The Formalisation of Protection Systems

submitted by

Ben Mankin

for the degree of Ph.D

of the

University of Bath

2004

Signature of Author ....

Ben Mankin

Summary

[JLS76] writes, “The Folklore is replete with stories of ‘secure’ protection systems being compromised in a matter of hours. This is quite astounding since one is not likely to claim that a system is secure without some sort of proof to support the claim. In practice, proof is not provided, and one reason for this is clear: although the protection *primitives* are apparently quite simple, they may potentially interact in extremely complex ways. Vague and informal arguments, therefore, often overlook subtleties that an adversary can exploit. Precision is not merely desirable for protection systems, it is mandatory.”

In this thesis, we construct a formalism which provides the required precision by showing that it is possible to construct a protection system which can be shown in short time to enforce a large class of organisational security policies.

We initially study the decidability of safety in specific protection systems using the common access-matrix model. We build upon this to the point where we can treat the systems themselves as computational elements and thus create a formalism which produces safety results for entire classes of protection systems.

Our study of the manipulation of rights encompasses a study of responsibility; we achieve a unification of the λ_{sec}-calculi for stack inspection with that for data inspection, and demonstrate an extremely close coupling between the calculi of responsibility and our formalism for decidability of safety in protection systems.

We use our formalism to model numerous existing systems and understand their strengths and their weaknesses; hence we are able to create powerful new systems without such weaknesses. Several such systems exist; we describe a sample implementation.

Acknowledgements

Thanks

With most sincere thanks to the many people without whom this thesis would still be a vague idea instead of a concrete reality.

I would like to thank my supervisor Julian Padget for his encouragement and interest in my subject and my father Raphael Mankin for reading it repeatedly and providing information and motivation from various security projects in industry.

Considerable effort was put into the proofreading of this thesis, especially by Jessica Bodman, who has been tireless in her pursuit of grammatical and tyopgraphical correctness, but also by Serge van den Boom, Mark Chappell, James Davenport, John ffitch, Will Lowe, Rayner Lucas (the 24-hour pedantry hotline), Antony Riley, Kathryn Rose, Wynke Stulemeijer, Frank van Waveren, and myriad others who had a hand, finger or fist in it at one time or another.

Carsten Führmann, Nicolai Vorobjov, David Pym and Richard McKinley helped me with the difficult bits with λ and x in them, and Leigh 'Blue' Caldwell gave valuable feedback from an industry perspective in the early days.

My sanity has been partially preserved[1] thanks to Martin 'Qaddafi' Brain, Alison Young, and the University of Bath Fencing Club, who let me beat people up when the stress got to me.

Dave Stewart and several innocent denizens of `#bblug` helped with the research on Vladivostok, and John Collomosse, Andrew Holt, Rachid Hourizi and Martin Owen led me down the winding garden path of backgammon. Or perhaps I led them.

As with almost any security thesis, there are Unmentionables™.

Apologies

With most sincere apologies to Lewis Carroll,[2] Tom Lehrer, Nicolai Ivanovich Lobachevsky, Augustus de Morgan, anybody I forgot to mention above, and jess.

[1] In vinegar.

[2] [Car60] had better appear in the bibliography if I don't want minors.

"It's very hard to talk quantum using a language originally designed to tell other monkeys where the ripe fruit is."

— Terry Pratchett, Night Watch

"Only be sure always to call it please, 'research'."

— Tom Lehrer, Lobachevsky

"First they came for the verbs and I said nothing, for verbing weirds language. Then they arrival for the nouns and I speech nothing, for I no verbs."

— Peter Ellis

"Verbosity leads to unclear, inarticulate things."

— Vice President Dan Quayle, 30th November 1988

Contents

List of Figures

List of Algorithms

List of Music

Notation and Symbols

Basic mathematics

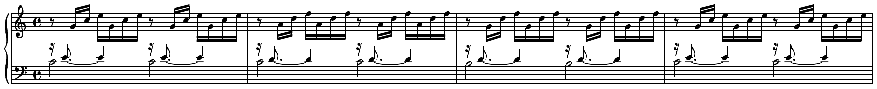
$\sum$	Sum.
$\bigvee$	Disjunction.
$\bigwedge$	Conjunction.
ϕ, ψ, ξ	Maps or homomorphisms.
σ, τ	Bijections.
κ, χ	Enumerations.
$\phi[A]$	The set $\{\phi(a) \mid a \in A\}$.

Computation and λ-calculus

λ	The lambda calculus operator.
β	A rule in the lambda calculus.
η	A rule in the lambda calculus.
ε	A fresh token in the lambda calculus.
δ	The transition function of a Turing machine.
Γ	The set of tape symbols of a Turing machine.
Ω	The divergent computation.
e, M	Arbitrary expressions in the lambda calculus.
$\mathrm{fv}(e)$	The set of free variables in an expression e.
$\mathbb{K}$	The domain of a tristate logic. $\mathbb{K} = \{\mathrm{true}, \mho, \mathrm{false}\}$
$\mho$	The value representing 'unknown' in a tristate logic such that $\neg\mho = \mho$.
$\mathbb{P}$	The set of all permissions.

Protection systems

Ψ	A protection system. $\Psi = (R, C)$
R	The set of generic rights of a protection system.
C	The set of commands of a protection system.
$\mathrm{cf}(\Psi)$	The set of all configurations of a protection system Ψ.
Q	A configuration of a protection system. $Q = (S, O, P)$, $Q \in \mathrm{cf}(\Psi)$
Φ	The empty configuration. $\Phi \in \mathrm{cf}(\Psi)$, $\Phi = (\emptyset, \emptyset, \emptyset)$
S	The set of subjects in a configuration of a protection system. $S = \{s_1, s_2, \ldots, \}$
O	The set of objects in a configuration of a protection system. $O = \{o_1, o_2, \ldots, \}$
P	The access matrix of a configuration of a protection system. $P : S \times O \to 2^R$
α	The name of access matrix command.
c	An access matrix command. $c \in C$.
r, ρ	Generic rights such that $r \in R$, $\rho \in R$.



1 Introduction

Many of us are familiar with the mechanics of breaking computer security systems. But while we may understand some of these procedures at a purely mechanical level, do we really understand *why* we are able to crack systems? Where do these vulnerabilities come from? What aspects of protection system design allow so many trivial misfeatures to become such major problems?

Many of us also have some knowledge of how to create a relatively secure system. Such systems tend to be minimally permissive, frequently to an extent that they become unusable. Modification of such a system by an untrusted user is unthinkable. But even the less "secured" systems we use for day-to-day work tend to disallow almost entirely the transport or reassignment of permissions, because there is a general lack of understanding of the dynamics of permissions.

Our lack of understanding of all aspects of protection systems has left us unable to answer a great many questions, including:

- "*What undiscovered vulnerabilities exist in a system?*"
- "*How can we design a system in which minor implementation errors in application software do not affect overall system security?*"
- "*How do we balance security against permissiveness?*"

All of these questions and many others can be answered by a formal analysis of the design of protection systems.

1.1 What questions can we answer?

So what questions can we answer? The questions proposed above have arisen as problems through limited experience with our current repertoire of protection systems. There are many more open questions in the field of protection. There are even some questions at the asking of which eyes are rolled skywards. Yet many of these questions are answerable.

We can build a system which automatically identifies vulnerabilities within itself.

This is not entirely a new concept; intrusion detection systems such as Tripwire ([SK03]) already have limited functionality for identifying vulnerabilities within a system. However, Tripwire and its ilk are concerned only with identifying the occurrence of *particular* vulnerabilities. The utility of such a system is based on a previous static and largely experiential analysis of a *sample* Unix configuration, with conclusions such as "*if the password file is writable to any user, then the system may be compromised.*" If the system configuration is modified in any way, both the previous analysis and the existing product are liable to be inaccurate or useless.

If we can efficiently analyse any configuration of a particular protection system, then the system can accurately advise of the consequences of any proposed configuration change before it is enacted. This suggested analysis is generally impossible. However, there exist protection systems for which it is possible in linear time and space, and we exhibit a selection of these systems in section 7.

We can build a system where mistakes in application code do not create security vulnerabilities.

Writing a *secure* program frequently involves considerably more than simply implementing the functional specification. If untrusted input can influence the program behaviour, then this input could cause the program to attempt a protected operation with a malicious outcome. The source of any such data must be considered by the protection system when a protected operation is attempted.

If the underlying system does not provide and maintain security information for input data, then it becomes the responsibility of the application to compute and maintain this information. It is a consequence of such a system architecture that security must be considered in every line of application code that manipulates untrusted input, thus making the system prone to security errors.

We can design a system architecture such that security information is automatically maintained by the system. This information will be considered by the decision processes of the protection system when necessary. In such a system, secure programs may easily be constructed by a programmer who is entirely unaware of the concerns of security. Security then becomes solely the domain of the protection system.

We can build a secure protection system yet still allow untrusted users to manage rights.

It is impractical in any large organisation for all protection system administration to be performed by a central system administrator. The size of such an organisation requires that control be delegated. However, the trade-off between flexibility and security in protection system design has always been biased towards the side of security, and consequently against this kind of flexibility.

Giving arbitrary users any freedom to administrate the protection system has always been considered too great a risk to security: If our protection system does not adequately protect itself from malicious modification, then a user might exploit this to destroy the system. Such a system would, of course, be fundamentally insecure, but we may not have any way of proving that.

We can build a system within which it is safe to delegate the administration of the system to the users, to the extent that the role for the central administrator in handling permissions is almost abolished. We can even establish such a system over a cross-organisational network such that each local administrator can administrate rights within his own systems, yet the systems work securely together as a network under a single protection system.

We can build practical, expressive systems with these properties.

Many of these properties will appear surprising to the reader familiar with traditional protection systems, and it is natural to wonder what is lost in order to make these gains.

Nothing is lost. What is gained is simply an understanding of what properties are held by various types of protection system. Instead of increasing security by making a system increasingly restrictive, we create security by design, by analysis and by understanding the mechanisms of protection. There exist many systems with desirable properties, and we may choose freely to use any of these systems.

1.2 What questions do we not answer?

We do not prove any implementation correct.

It is important to make the distinction between correctness of design and correctness of implementation. Even given a correct design, any implementation containing an error may be insecure, but such an error may be corrected to create a secure system. A system with a flawed design may not necessarily be corrected; it must often be redesigned from scratch. Our work includes the design, example algorithms and complexity results for these algorithms, but does not include any technique for proving an implementation correct.

There is security, and there is insecurity.

A system which can be broken is not secure. A system which cannot be broken is secure. A system which we do not yet know how to break, but has not been formally proven secure is not to be considered secure. The common phrase, "*more secure*" must be interpreted as, "*more likely to be secure*". This is not a work on probability, and so we will not use such phraseology.

1.3 Our Approach

In this thesis we build an understanding of protection. We have learned many things about protection in the real world through thousands of years' experience. Many concepts with which we are familiar may be identified in our box and key example, and eventually will form the building blocks of a formal model. However, while we later provide a considerable amount of formal material, it is more important to understand the fundamental concepts and have an appreciation for how the models actually work.

1.3.1 Protection

We first introduce the concept of the protection system in section 2. We describe the shape and operation of a protection system in the real world, the example from which all computer protection systems are derived, that of a box with a lock and key, presented in section 2.1.2. It is from these roots that we discover the mechanisms of access control, or protection.

Many of us are already familiar with two stages of an access control decision: authentication and authorisation. It might come as a surprise to learn that every access control decision actually consists of *three* distinct computations, described more fully in section 2.1.3. First, we must identify the people around us, the authentication stage. Second, we must work out which of these people is responsible for making the request; this person is the "current principal". Third, we must decide whether that person is permitted access. The latter two stages form what has previously been called the authorisation stage, but they will rapidly become clear as distinct, but intertwined mechanisms. Of these stages, we leave authentication to the cryptographers, and study only the elements of authorisation, or as we call it, '*protection*'.

Having identified the mechanisms, we must have the naming of parts. Some of the basic definitions which follow in section 2.2 and the design choices of section 2.3 may appear obvious to the modern reader, but they are not automatic, and must be explicitly codified as axioms.

1.3.2 Policy

Our objective is to achieve security. Security is a funny thing: We might glance at a system and pronounce it apparently '*secure*' without any measure of objectivity, but when it is broken, we exclaim, "*Oh, it must have been insecure!*"[3] We do not yet have an absolute concept of security. By contrast, the term "*secure*", as we apply it to protection systems, must be an absolute. A protection system is either secure, or it is not. But there is as yet no yardstick by which we may classify a protection system as secure or insecure. What are we protecting, from whom, and against what?

Section 3 both introduces and completes our study of policy. A policy is an arbitrary specification for the protection system: any protection system which satisfies a policy is secure, or correct with respect to that policy. A policy may state who must be allowed access to objects, and who must be denied access. The protection system is responsible for enforcing the policy. But a policy makes a far stronger requirement than the specification of a state of a protection system. The state of a protection system indicates the accessibility of objects at a given time. A policy states what the accessibility of objects must be *at all times*, even into the infinite future of the protection system. It is this *at all times* which defines "vulnerability". If a protection system can, in the future, allow access, then it is considered to be vulnerable, and we call the system "*unsafe*". Otherwise, the system is "*safe*". Thus policy is the yardstick used for identifying vulnerabilities, and provides the questions we would ask of any analysis of our protection system.

[3] But we've fixed it and it's secure now, honest!

Having provided that important motivation and element of understanding, the policy steps into the background of the thesis. We will also prefer not to use the word “secure”, since an absolute definition in terms of policy is available.

1.3.3 Formalisms

A basic formalism for protection systems has been around since 1976; we reproduce it in section 4. Part of the reason for the longevity of this model is its expressiveness; it can express formally the behaviour of any protection system. Such expressiveness is also its downfall: it can simulate a Turing machine, and therefore no useful positive results about protection systems may be constructed from this formalism. The major negative result of this formalism is reproduced in section 4.4; it is the undecidability of the “General Safety Problem”, which states that it is not possible to analyse an arbitrary protection system to discover if it is safe.

The historical presentation of the formalism for protection systems is unfortunately lacking in at least one major respect, the consideration of the “current principal”. While this does not affect the historical results, it does affect our ability to use the formalism in any practical analysis, and we must correct this omission in section 4.6 before we can apply the formalism to practical systems.

1.3.4 Mathematics

We may also construct many useful tools for manipulating protection systems as computational entities. Such tools allow us to classify a system, to describe the expressiveness of a system, and to make transformations of a system while retaining the properties of that system. It is these tools upon which we will rely to show that a result proven for one system transfers to another *equivalent* system or that a protection system is as *expressive* as another system. Our definition of equivalence is especially important, since it allows us to identify and study equivalence classes rather than individual systems, thus saving considerable effort.

Section 5 is a collection of this formal work, in which we introduce the full definitions of simulation, equivalence and expressiveness. It is frequently these definitions which allow us to make objective statements about the utility or behaviour of protection systems. Rarely will we use these definitions formally, although we frequently make implicit use of them by reference.

1.3.5 Classes

Given a protection system, any decidability result for that system will hold for the equivalence class of systems containing that system. However, we can also prove results about classes larger and more general than equivalence classes. A sufficient restriction of the general formalism for protection systems might produce a class of protection systems such that the “Class Safety Problem” is decidable. If any class safety problem for a particular system is shown to be decidable with sufficiently low complexity, then an algorithm for deciding the safety of particular system configurations may be constructed for the system. Thus we may identify protection systems which are correct, expressive, fast and decidably safe in low order polynomial time. Such systems may be of practical use.

In section 6, we study a number of classes of protection system strictly within the formalism. But the formalism itself does not provide any method for constructing classes worthy of study. The work in this section is arduous, and consists mostly of laborious stabs in the dark which fail to hit anything.

There is another method by which we may identify a class of practical protection systems. Section 7 shows that we may also construct a practical protection system from the ground up. We take first the simplest possible system and show that the safety problem for this system is decidable. We then extend it to satisfy many desirable functional requirements, and thus by a process of exploration, identify a useful protection system with a decidable safety problem. We study some instances of the class safety problem explicitly in this section, but we focus on the practical and administrative functional requirements for our '*ideal system*'.

It is indeed possible to produce an ideal system in this way, and produce elegant and fast algorithms to show that such a system is safe. However, in the development of this system, the business requirements have led us to create a far more powerful structure of permissions than we had previously anticipated, and these give some surprising results. We also discover towards the end of section 7 that the concept of the '*principal*' is far more complex and important than previously realised.

1.3.6 The Current Principal

Of our two initial problems of protection: identification of the current principal, and access control, we have traditionally considered computing the first correctly to be trivial, and deciding the safety of the second to be more difficult. After all, the current principal is just the person making the request. But section 7 inspires our interest in the principal as an abstract computational entity. Also, all the mechanisms that we have traditionally used for identifying the current principal appear to be flawed! We can not simply assume that the person making a request is '*responsible*' for the request because it isn't generally true.

We learn in section 8 that the principal is an abstract object not necessarily representing any person. We may represent any principal as a set of permissions and thus compute with principals to identify a current principal. By performing this computation correctly we may actually make the access control decision trivial; either the current principal has the required permission, or it does not. But now, identification of the current principal is the hard task. We show, in one of our more formal sections, the weaknesses of the various methods of computing a current principal. We can also show that there is really only one "right answer", only one correct way to compute the current principal, if we are to create protection.

1.3.7 Case Studies

The case studies are fairly informal with respect to much of what has gone before. It is sufficient to classify a system by the model it attempts to implement; usually this is clear, but sometimes more justification is needed. The case studies make two major illustrations. First, they show that the results from our formal study match experiential evidence from the real world. Where we predict a hole there exists a hole. Second, they show how little has been done to exploit the possibilities that a better protection system design has to offer. The author has twice implemented such a system in a medium scale environment, but of our case studies only two allow the transport of rights by unprivileged users. Of those, one is obsolete and one is the author's own implementation.

1.3.8 New Applications

If we use the optimal structure for permissions developed in section 7 and the mechanisms for the identification of the current principal from section 8, we can achieve a surprising system which retains all the positive results from our previous work. An informal sketch of such a system which builds on our previous work is presented in section 10, and we hope that this design can form a basis for future implementations of protection systems.

Section 11 contains our conclusions and section 12 describes possible further work.

1.3.9 Previous Work and Bibliography

A list of reference materials may be found in the bibliography on page 221. We reference work by other authors throughout the main text, rather than grouping all references in a single section or subsection dedicated to "Previous Work". In this way, references will be placed alongside the material to which they are relevant.

We also provide an index of concepts on page 229.

1.4 The Organisation of this Thesis

1.4.1 In Pictures

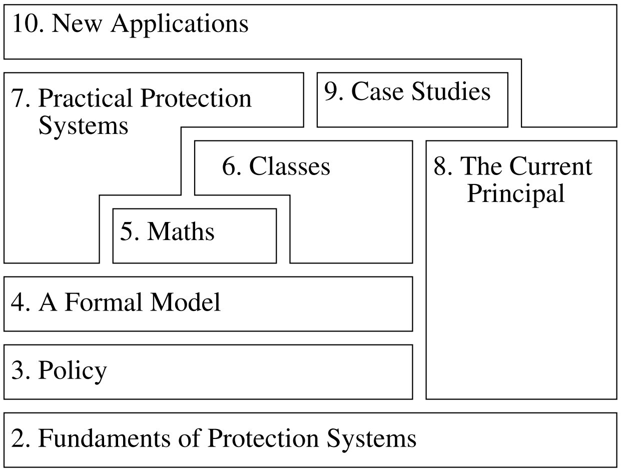


Figure 1: The structure of this thesis

Figure 1 is a dependency diagram of the thesis. We build upwards from the fundamental ideas to the new applications. Any section which rests on a lower section depends on the material within that section. Thus it is possible to study section 8 without understanding the material in section 5. However, it is recommended that the sections be read in order, since later sections solve problems inspired by earlier ones.

1.4.2 In Words

In section 2, we introduce the concepts of security and protection and introduce our first model of a protection system. We use this first model to help us identify the basic mechanisms of protection. We make some essential definitions, and establish much of the groundwork for a fuller formalisation of the mechanisms of protection.

Section 3 introduces the concept of a system policy: those requirements made of a protection system by the system administrator, and will motivate our concept of "*adequate*" protection by establishing a hard set of requirements for any protection system.

In section 4, we introduce the formal model and the "*General Safety Problem*". We introduce with these many concepts and tools with which we may work. Some major flaws are exhibited in the previous study of this problem, most notably in section 4.6, where we extend the formulation of the safety problem with the introduction of the concept of the current principal.

Section 5 demonstrates techniques for manipulating the definition of a protection system, including the important definitions of simulation, equivalence and expressiveness.

Section 6 uses these tools to identify subclasses of protection system for which the safety problem is decidable. We will endeavour to identify subclasses large enough to be generally useful and contain security models currently used but small enough to be decidably safe in low order polynomial time. This section includes both new and existing proofs.

Section 7 takes our theory back into the real world. We build a restricted taxonomy of protection systems designed to include many common real world systems. We compare the systems from this taxonomy using both quantitative and qualitative requirements to discover some extremely powerful and versatile systems with security decidable in linear time.

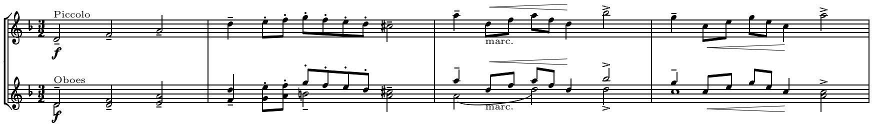
In section 8, we will show that we can reliably define and identify a current principal at any point in a computation. We formalise many models in a single λ_{sec}-calculus, and show many equivalences and relationships between this model using the common formalism.

In section 9, we will use the knowledge gained from our formalisms to produce full case studies of various existing systems, showing how they fit into our model and where they are lacking. In this way, we will expose the vulnerabilities in these systems and show how they may be broken or *hacked*.

If we can understand why existing models are lacking, then we can design new systems without such faults. Section 10 sketches how we might extend traditional algorithms up to and including dealing with a network of untrusted hosts.

Our conclusions are in section 11, and some notes on further work are in section 12.

But first, we must meet the cast . . .



2 Fundaments of Protection Systems

There is much preliminary material which must be established before we may easily understand the formal work which follows. In this section, we seek to give an overview of the elements of our formalism, establish some convenient definitions, and summarise the design choices and requirements developed from the experience of the last 40 years of operating system design. Many of the definitions in this section are of a more general form which will apply throughout the thesis; in places where necessary, a more specific definition may be made later. Little in this section should take the reader by surprise.

2.1 An Introduction to Protection

Jerome Saltzer and Michael Schroeder wrote a paper entitled, "*The Protection of Information in Computer Systems*" ([SS75]). The accuracy of the paper with respect to modern systems is remarkable in such an early work, especially given that relatively few examples of real world protection systems were available to the authors; the paper is contemporary only with such systems as CTSS ([Cri65]), the DECsystem/10 ([Sto70]), ADEPT ([Wei69]) and TENEX ([B$^+$72]). The following definitions are made in the introduction to that work.

Definition 2.1 (Security ([SS75])). With respect to information processing systems, used to denote mechanisms and techniques that control who may use the computer or modify the information stored in it.

Definition 2.2 (Protection ([SS75])). (1) Security (q.v.). (2) Used more narrowly to denote mechanisms and techniques that control the access of executing programs to stored information.

Definition 2.3 (Protected Subsystem ([SS75])). A collection of procedures and data objects that is encapsulated in a domain of its own so that the internal structure of a data object is accessible only to the procedures of the protected subsystem and the procedures may be called only at designated domain entry points.

Definition 2.4 (Protected Object ([SS75])). A data structure whose existence is known, but whose internal organisation is not accessible, except by invoking the protected subsystem (q.v.) that manages it.

2.1.1 The Motivation for Protection

The very first computers were single-user single-task systems. Any data or code in the store of the system was related to that one task and provided by the one user. As these computer systems became simultaneously accessible to multiple users, it became increasingly important to control access to the resources of any shared system in order to prevent theft, denial of service, and damage to data and property.

Protection systems were first developed as a consequence of the introduction of time sharing; they were designed to prevent faults in a program or in the system from affecting other programs executing on the system. The simplest of these early mechanisms were base and bound systems, and these were rapidly followed by hardware virtual machines ([SS75]).

Timesharing, and later networking, rapidly brought about a drastic increase in shared computing resources. Whereas the motivation for protection had been to prevent accidents, it became increasingly important to protect computer systems and data against malicious acts. It became increasingly less likely that users of a computer would be working cooperatively, and thus it became important to prevent any user from reading, copying or destroying another user's data, or from denying another user access to resources ([Lam71]).

Today, the majority of computers are accessible to multiple users. Most computers are connected to a network, and the internet allows millions of users to submit arbitrary requests to almost any networked computer.[4] However, the data and resources of such computers are not necessarily intended for public use. It is not desirable, for example, that the records stored at a credit card bureau be accessible to any user who cares to submit a request. It is therefore

[4] An unsupervised teenager with a modem is as dangerous as an unsupervised teenager with a gun.
— Gail Thackeray, Arizona Assistant State Attorney

critical that a computer system be able to differentiate correctly between those requests which are to be permitted and those which are not to be permitted: those actions which are desirable to the owners and maintainers of the data, and those actions which are not. The principles and practice of '*protection*' have grown up around this need.

Protection itself is a subfield of security, and is concerned with classifying access requests into those which are desirable and those which are undesirable, and correctly enforcing such a decision. The term '*privacy*' is usually used to refer specifically to the protection of information from undesired reading or dissemination.

In practice, the type of action which the protection system is designed to control has little impact on the correct design of the system. The protection system considers only '*actions*', and permits or denies them according to some set of rules, called a '*policy*' (section 3) set by the system designer. In order to account for all possible complexity, the formal model of a protection system introduced in section 4 still considers the object and the action to be performed separately. Even our definition of a '*permission*' in definition 2.9 treats the object and the action separately. It is not until the proof of theorem 6.8 that we formally show the concept of 'action' to be unnecessary.

We do not specifically study privacy since it is a subclass of protection concerned only with the action of '*reading*' or passing on data to parties who will read it, and it will be clear how our results may be used to achieve the same for privacy as we achieve for more general protection.

It is important to remember that the purpose of the design of a security system is not to create a system without vulnerabilities, but to create a system without the opportunity for vulnerabilities to be created. If we can achieve this larger goal, then the applications and even the systems programmers are free to concentrate on the desired functionality of the system without further explicit concern for security.

2.1.2 A Model for Protection

Example 2.1 (Boxes and Keys). A principal wishes to keep a number of objects safe. He protects each object by building a box with a lock, putting the object into the box and only giving a key to this box to trusted principals.

Any person wishing to access a particular object must hold the key to the box containing it. The key to a different box will not do, nor may the person open the box with a correct key which another person is holding; he must use his own key. Possession of a key is ten tenths of the law.

Example 2.1 is the simplest possible analogy for the protection of an object.[5] The object in the locked box is the protected object of definition 2.4. However, we have already extended our conceptual library with the idea of the '*principal*' and the '*key*'. Without the correct key, a principal may not open a particular box. Without the relevant privilege, a principal may not access a particular protected object.

[5] Thus conveniently satisfying the criteria of *economy of mechanism* and *psychological acceptability* from [SS75].

In the real world, there are other ways to access the object. Given the appropriate tools, smashing the box or picking the lock will suffice. Subverting a person who holds a key might give indirect access. Finding the design for a key and duplicating it by mechanical means might also work. Many of these methods have parallels in the world of computing: physical attacks on the system hardware, social engineering and password guessing or cracking are some of the most common methods of security compromise ([Sch96]). However this represents a considerable extension of the problem of protection, and under most circumstances, most of these extensions are not applicable. Our material is concerned only with the simplified model of principals, keys, boxes and any relationships between them. Our formalisms will model only this pure world.

2.1.3 The Basic Mechanisms of Protection

A protected object may only be accessed through the protected subsystem managing it. This protected subsystem may elect to permit or deny access to the object. It is responsible for emulating the lock on the box. Already, with the help of the definitions of section 2.1, we have identified the major participants in the model. We must now understand how these participants interact if we are to understand the mechanisms of protection within our model.

According to our model, a principal may access an object if and only if he owns an appropriate key. It is the responsibility of the protected subsystem to identify the acting or '*current*' principal and then decide whether this current principal owns an appropriate key for access to the requested object. In order to model this interaction, the protected subsystem must therefore be able to answer reliably and correctly the following questions.

1. **Which is the currently active principal s?**
2. **May a given principal s access a given object o?**

The techniques used to answer these two questions may be considered almost entirely independently. The input to the first question is some *state* of the system and the output is some conceptual "*current principal*". The input to the second question is the object being accessed and the principal identified by the answer to the first question. A pair of algorithms for answering these questions establishes a protection system.

It is important to make the distinction between the problem of *authentication*: identifying a principal external to the system making a request into the system, and the problem of identifying which of the authenticated principals is making the current request. The second of these problems is a part of the authorisation mechanism. A more complete flow diagram for a security system would include the authentication of principals, as illustrated below.

$$\begin{array}{c} \text{Authentication} \\ \text{of principals} \end{array} \longrightarrow \begin{array}{c} \text{Identification} \\ \text{of current} \\ \text{principal} \end{array} \longrightarrow \begin{array}{c} \text{Authorisation} \\ \text{of request} \end{array}$$

In this thesis, we are concerned only with authorisation; that is, protection. We assume that all principals in the system have been identified prior to entry into the protected subsystem.

The problem of authentication is covered extensively in texts such as [MvOV96] [Sch96] [FS96] and [Pfl97]. Our base assumptions include that all principals acting within the system have been correctly authenticated and that we must merely decide which one is the *current principal*.

2.1.4 From Static to Dynamic Systems

In example 2.1, the assignment of the keys to principals was fixed by design, and from this example, we derived a theoretical model where the key assignment was a fixed part of the system, thus establishing a "*static*" protection system. The mechanism described in section 2.1.3 consequently satisfies all the requirements for protection in a static system, a protection system where the access specifications never change. Systems like this are remarkably common; they may be established by some box-builder and left to run forever without modification. However, these static systems tend to be simple systems like firewalls with rules like
"*Everyone inside the firewall may access anything, anyone outside may not access anything.*"
Many large distributed mobile systems may reasonably use a static protection system. For example, a mobile phone network has a large, geographically distributed userbase using diverse hardware and software; but within such a network, the access specifications remain static: the telephone company may change charging structures, access information regarding users, and so on, while users may only make calls.

A static protection system is not practical for all environments. Requirements change. New objects are introduced and old objects removed. Whole divisions may be formed within a company, and whole divisions may be sold or liquidated. Any change in the structure of the organisation using a protection mechanism may require a change in the configuration of the protection mechanism in order that it may continue to discriminate between desirable and undesirable access.

It is unreasonable to build a new static protection system to allow for each change, therefore we must permit changes to an existing system, that is, make it dynamic. We must add to our example a set of rules controlling the copying, and possibly the destruction of keys, and from this we must develop an extension to our mechanism for deciding an answer to the following question.

- **May a given principal s change the access specification of an object o?**

It would be premature at this stage to presume that each element of key assignment information is itself a protected object and modifiable under similar conditions to any other object in the system. However, it is quite natural to consider the possibility, and we will discover in section 7 that in many of the better systems, it is in fact so.

2.2 Basic Definitions

In order to understand and manipulate a formalism, we must establish some *feel* for the elements used to construct it. The basic definitions presented in the "*Glossary of Computer*

Security Terms” ([Cen88]), infamous as the “*Aqua Book*”, form a significant part of the standard vocabulary of the security community, but they are extremely terse and give little insight into the objects which they describe. Even the excellent definitions in [SS75], to which the reader is recommended, are the product of practical experience and may not immediately be understood without the benefit of that experience.

We will therefore present our definitions as self contained wholes. The reader is likely to be familiar with most of these definitions, but our explanation of each definition may throw more light on the work to come.

2.2.1 The Trusted Computing Base

Definition 2.5 (Trusted Computing Base ([Cen88])). The *Trusted Computing Base* or *TCB* is the totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy.

A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to enforce correctly a unified security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user’s clearance level) related to the security policy.

In the context of dynamic protection systems, it is useful to understand the concept of the TCB in order to understand which elements of a protection system may be modified, and which may not. The protected subsystems of definition 2.3 must themselves be protected from arbitrary modification. They may be entirely immutable, or as also suggested by [SS75], they may protect their own implementation. What matters is that:

> No principal may modify any protected subsystem in order to change the access specifications in a way that he would not have been able to do through the intended mechanisms of the protection system.

This is verbose, and is frequently abbreviated for convenience, comprehensibility and readability to “*Nobody may modify the TCB.*” or “*Nobody except the system builder may modify the TCB.*” which is clearly not the same, but serves for some practical purposes. Usually the TCB is built when the system is initially designed and considered inviolable from then on.

In example 2.1, the boxes and the static key assignment information constitute the TCB. The rules dictate that only the builder of the boxes may violate the boxes themselves. Everybody else must hold a key to access the box. Keys may not be arbitrarily copied (copying of passwords and encryption keys is outside the realm of this document), they must be assigned to a user at the time of the construction of the TCB. The entry points into the TCB will also be well defined. The action of putting a key into a lock to open a box is an example of an entry point into the TCB, causing the TCB to release the contents of the box to the current principal.

When we define a protection system, we will consider the system itself to be in the TCB and hence inviolable. We will also assume that the TCB is responsible for actually executing the rules of the protection system and that this execution may not be influenced by any (possibly malicious) principal. We do not explicitly use this definition of the TCB; it serves only to give an understanding of the inviolability of the protected subsystems surrounding objects.

2.2.2 Objects

Definition 2.6 (Object). An object is an entity which may be accessed.

Our definition of *object* is really an abbreviation of definition 2.4. It is always implicit in our work that accesses to an object are intercepted by a protected subsystem which may make an access decision. Objects are otherwise opaque. We care nothing about the structure or contents, and we will use them only as labels or names in order to study the protected subsystem surrounding them.

2.2.3 Principals

Saltzer and Schroeder ([SS75]) make a very perceptive definition which we see no need to replace.

Definition 2.7 (Principal ([SS75])). The entity in a computer system to which authorisations are granted; thus the unit of accountability in a computer system.

A principal is an entity which performs operations and may be held responsible for them. A principal may correspond to some real person or to some entity representing joint responsibility between multiple simpler principals. The concept of *responsibility* is an important one, but computation of responsibility did not feature in security literature until comparatively recently. The concept does not take centre stage in this thesis before section 8, where we learn to construct principals to represent arbitrary designations of responsibility. Until that time, this definition will remain a little vague and informal, since we have little motivation for a more precise definition of an object which we have not yet attempted to manipulate.

2.2.4 Privileges

Definition 2.8 (Privilege). A privilege is a triple (s, o, r) where s is a principal, o is an object and r is a token from some finite alphabet R. r is sometimes termed a *right.* Privileges may be created, modified or destroyed only through the mechanisms of the protection system. They may not be forged.

This definition formally expresses the notion from example 2.1 that a key only works for the person who *owns* it by binding both the identity of the owner and the identity of the object into the privilege. A privilege may only be used by the principal identified within it. We will not often have occasion to be so formal.

An object may have several entry points, each of which opens to a different kind of key. Any given entry point on a given object may be accessed using the appropriate privilege. ('*Fred*', '*The safe*', '*open*') is a particular type of real-world privilege, usually instantiated as a key in

Fred's pocket, allowing Fred (and only Fred) to open the safe. (`root`, `/etc/passwd`, '*write*') is a privilege frequently found on Unix systems.

We assume without loss of generality that if a privilege (s, o, r) exists, it is unique. Were multiple copies to exist, they would be indistinguishable and redundant. Removal of the privilege from the system would be assumed to remove all copies. Therefore, the total number of privileges in the system is polynomial in the number of principals and objects in the system.

Definition 2.9 (Permission). A permission is a pair (o, r) where o is an object and r is a right.

The permission is related to the privilege. We grant permissions to principals, and in doing so, we create a privilege. Our definition of a permission is derived directly from this usage, and corresponds to the formal definition of a capability in [DvH66], the more intuitive definition of a permission from [SS75] and many newer works. We rarely, if ever, meet concrete permissions as they do not form a part of our model; the concept alone is required.

2.2.5 Superusers

Definition 2.10 (Superuser). A principal s is a superuser if and only if (s, o, r) is a privilege for every o and for every r in the system.

A *superuser* is a principal with every permission in the system, and is therefore able to access any object at will, and by axiom 2.1 to come on page 18, to modify any *modifiable* part of the protection system. A superuser may not modify non-modifiable parts of the protection system or the TCB.

In example 2.1 on page 12, the locksmith responsible for building the boxes may be a superuser since he is at liberty to keep a key to each box he constructs. If he discards any keys, he is no longer a superuser. In practice, a system is not restricted to one principal which is a superuser, nor is it required to have any superuser.

Occasionally the term *superuser* is misused to refer to "*the highest authority which is instantiated in an actual principal*". This is incorrect. The concept is uniquely defined by having *all* permissions. We will extend this idea in section 2.3.7, where we explain that *all* principals are identifiable only by the privileges they hold. It is therefore sufficient and correct to refer to "*the superuser*".

2.2.6 The Current Principal

We met the concept of the current principal informally in section 2.1.3, where we referred to the active principal in example 2.1. In that analogy, the current principal is the person holding a key and attempting to open a box. We may now make this a little more formal.

Definition 2.11 (Current Principal). The *current principal* is the principal responsible for the execution of the current operation.

The designer of a protection system will frequently intend that a principal p be allowed to perform an operation but that some other principal, say p', be prevented from performing the same operation. The principal '*performing*' the operation is the '*current principal*', and the access decisions made by a protection system must depend on this current principal, otherwise all decisions will either "allow access to everybody" or "deny access to everybody". A protection system is therefore required to identify the current principal and perform permissions tests with respect to the permissions of that current principal.

In contrast with our intuition from example 2.1, this definition does not imply that the '*current principal*' is some atomic entity, a person, a process or an object. We hinted at this idea in section 2.2.5, but it is not explored fully until section 8. We also lack formal definitions of both '*responsibility*' and '*performing*' until definition 8.4 in that section. However, the definition above is sufficient for our purposes until then.

2.3 Design Choices

A protection system which allows arbitrary access decisions would be very hard to formalise, and any formalism developed from a class of such systems would be difficult to study. Therefore we will make some preliminary design decisions about what type of protection systems we will study, thus to restrict our formalism.

History has taught us that there are some choices which are "a good idea" to make, for reasons including simplicity and fault tolerance. Most of the decisions in this section will appear "obvious", certainly to the reader already familiar with protection systems in practice. Some of the following choices are made as axiomatic decisions with some justification. Still others may be made without loss of generality in our protection system and are suggested at this point to simplify the introduction of the formalisation of protection systems later.

2.3.1 Positive Tests

First, we axiomatise a statement made in about 1965 by E. Glaser,[6] who proposed that "*the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted.*" ([SS75], considerable treatment in [Den76]).

Axiom 2.1 (Positive Tests). *The protection system may test only for the presence and not for the absence of a right.*

Our intuition is satisfied since one may only open a box in the real world if one holds the key. There does not exist a lock which will only open if one does not hold a specific key. Correspondingly, in a computer protection system, it is necessary to show that a user holds (or can acquire) a permission in order to access an object. There is no access "by default". A protection system may then be considered to be a proof system within which one must

[6] Some of the seminal issues were widely spread by word of mouth rather than by publication, and hence references are unavailable. We attempt to identify the individual responsible for proposing ideas wherever possible. This concept was published in [DvH66], who said, "*A computation must be denied access to memory words and other objects of computation unless access is authorised*".

prove insecurity of a protected object with respect to a principal using the privileges currently present in the system in order for that principal to be allowed to perform a protected operation on the object. Usually these proofs will take a fairly standard form and so their construction can be automated. This form is usually a search over the rights of the principal, but in some more complex systems, such as those in section 7, the algorithm may be correspondingly more complex.

2.3.2 Compound Objects

In the framework of example 2.1, multiple objects in a single box are not separately protected, and are indistinguishable from the point of view of the protection system.

Axiom 2.2 (Compound Objects). *If two objects are protected by the same permission, then they are indistinguishable from the point of view of the protection system and may be considered as a single object.*

Our formal definition of the permission makes this definition a tautology. However, given a more intuitive definition of a permission, this statement is necessary.

If the operation to be performed on an object has no effect on the security decision process of the protected subsystem, then we may ignore this ‘operation’ argument for the purposes of computations regarding safety and consider all accesses equally. Equivalently, if a box contains more than one object and the choice of object being accessed has no effect on the security decision process, then we may consider all the contained objects as one.

Conversely, if the operation or target object does affect the security decision process, then we may consider the pairs $(o, a) \in \text{objects} \times \text{operations}$ as separate objects in separate boxes. The protection system is not concerned with what is done with the protected objects, but only with control of the access points.

2.3.3 Self protection

In any dynamic protection system, there will exist some mechanism for modifying privileges. This mechanism must itself be protected. We choose to be concerned only with protection systems which are *responsible for protecting themselves*. This is a justifiable axiom.

Axiom 2.3 (Self Protection). *The protection system must be responsible for protecting itself.*

An alternative and equivalent formulation of this axiom is the axiom of closure or completeness.

Axiom 2.4 (Closure). *There must be no entity outside the protection system which is able to modify the system.*

Any solution which protects a protection system using another ‘controlling’ protection system is unsuitable for analysis since arbitrary changes might be allowed by the controlling system. We must at least attempt an analysis of the controlling system, since any insecurity in the controlling system will also create an insecurity in the controlled system. Any analysis of the controlling protection system will reintroduce the problem which we are trying to solve for the base system (no significant real world OS examples are known but [SCFY96] achieves this with fine style), therefore we win nothing from an analytical perspective by protecting a protection system with another protection system.

We might alternatively consider the two systems as one larger combined system and analyse the whole, but in this case the system becomes self protecting, and our axiom holds. Thus the axiom is justified.

Many simpler self protecting systems have a discriminated privilege which allows maintenance of the protection system. Any principal holding this privilege may then change access specifications arbitrarily. No other principal may change access specifications. This privilege usually confers effective superuser status on the principal holding it since such a principal may grant itself every permission in the system. From a purely qualitative point of view, these trivial systems are inappropriate for a large dynamic or distributed network. While they are undoubtedly secure,[7] they are not practical, since it is impractical in any changing organisation to have one person who must perform every privilege modification, however trivial. We wish to stretch the bounds of what is decidably secure in the direction of what is practically useful.

2.3.4 Representation of Privileges

Privileges are protected, and may not be modified (see definition 2.8). The only operations which may be performed on privileges are creation, deletion and test for existence. There are many possible representations of privileges which allow for these operations. Some of these representations are *explicit*, in that they explicitly store a set of privileges which exist in the system, but such a set may be extremely large to the point of becoming impractical to store. Some systems, such as Java ([Gon03]), use an *implicit* representation of privileges. An implicit representation of privileges is a mechanism providing an interface to a hypothetical or implicit set of privileges in such a way that the basic operations may be performed on this implicit set. Since an implicit representation does not necessarily store a set of privileges, it may conveniently and efficiently represent an unbounded set of privileges.

While the formalism in section 4 specifies an explicit representation of privileges, the privileges are only manipulated through the interface allowing creation, deletion and test for existence, and thus none of our work will depend on the representation chosen.

We do not generally assume that the set of principals or the set of objects is bounded, therefore it is not generally true that the set of privileges is bounded. In section 10, we see that it is not even necessary to assume that these sets are finite.

2.3.5 Properties of Explicit Representations of Privileges

If we do choose to use an explicit representation of privileges, then there are three immediately obvious choices.

[7] If you trust your superuser, then surely nothing can go wrong!

- **Capabilities:** Each principal stores a list of privileges which it may use at will to access objects. A principal will select the appropriate privilege from its list when it attempts to access an object. The protection system must identify the current principal and verify that an appropriate privilege has been presented before allowing access by checking the requested object and the current principal against the given privilege.

  Examples of this mechanism include capabilities ([SSF99]) and certificate based systems ([Sch96]).

- **Access Control Lists (ACLs):** Each object is associated with a list of privileges which may be used to access it. The protection system must identify the current principal before knowing which privilege to use.

  ACLs are frequently implemented in database servers and are specified in the POSIX specification for Unix ACLs ([PASC97]), implemented in Linux ([Ano99]), Solaris, HP/UX and others.

- **Access Matrix:** The privileges may be stored in a global table. (see figure 2 to come on page 36) Again, the protection system must identify the current principal before knowing which privilege to use.

  This mechanism is less common, but examples include the MySQL database server ([MyS03]).

In every case, a relationship between principals and permissions is established such that it is possible in little time to state for a given principal s, object o and right r, whether s has right r over o, that is, whether a privilege (s, o, r) exists. Thus the required interface is always provided.

The difference between the explicit representations becomes more important when the protection system is asked, for the purposes of accounting or analysis, to enumerate all principals which may access a given object or all objects which may be accessed by a given principal.

- In the case of capabilities, identification of the set of principals which may access a given object o involves a linear time search of all principals.
- In the case of object ACLs, identification of the set of objects which may be accessed by a given principal s involves a linear time search of all objects.
- In the case of an access matrix, both operations may be performed in constant time by reading the appropriate row or column from the access matrix.

All of these algorithms are executed in polynomial time. More importantly, it is possible to transform between these representations in polynomial time and it is possible to maintain all three representations simultaneously for only a constant factor of overhead. Thus any algorithm for deciding safety which relies upon performing these enumerations may execute in polynomial time.

2.3.6 Properties of Implicit Representations of Privileges

The choices for an implicit representation mirror those for the explicit representation: the same three possibilities exist. The relationship between principals and objects objects specified by the existence or nonexistence of privileges (s, o, r) for any right r may be considered as is convenient by functions

$$k_{\text{permissions}} : S \to 2^O$$

the permission map from a principal to the set of objects over which it holds access permissions,

$$k_{\text{acl}} : O \to 2^S$$

the ACL map from an object to the set of principals on the ACL or

$$k_{\text{matrix}} : S \times O \to \{\text{true}, \text{false}\}$$

a global matrix specifying whether a given principal has a privilege over a given object. The same properties hold of these maps as held for the explicit representations.

2.3.7 Representation of Principals

The map $k_{\text{permissions}}$ motivates a representation of the principal as a set of permissions. We justify our future use of this representation by defining

$$s = \{(o, r) \mid (s, o, r) \text{ is a privilege}\}$$

in other words, the principal is identified only by the rights held by that principal. $s \in 2^{O \times R}$. As a corollary, two principals with identical privileges are indistinguishable. We hinted at this corollary in section 2.2.5 when we stated that two superusers are indistinguishable.[8]

The representation of a principal as a set of privileges may be used as an axiomatic definition in computations of responsibility ([FG02], [PSS01]). Later, in section 6.4, we will show that a protection system with only one right is equivalent to a protection system with many rights; therefore it will be possible to ignore r and let $s \subseteq 2^O$, justifying this axiomatisation.

We will use this representation extensively in material following from our computation of the current principal in section 8, but we do not allow this representation or its corollary in introductory work on protection systems and safety following from section 4. Until that point, it is sufficient to ignore the question of representation, and consider a principal only as defined in definition 2.7.

[8] It is not appropriate to make a similar definition expressing objects in terms of principals, since objects contain data, and this distinguishes them in operational terms. The principal exists only to support the protection system, thus we may do with it as we will.

2.4 Design Considerations

There are many properties of a protection system which are not required to provide protection, but are desirable because they make the system easier to design, easier to operate, or provide other desirable functional requirements not directly related to security. These requirements will not feature in our formalisms, but will return when we start to design our "ideal systems" in section 7.

2.4.1 The Principle of Minimal Privilege

Peter Denning stated what we will call the "Principle of Minimal Privilege" as an element of "Error Confinement" in his paper entitled "Fault Tolerant Operating Systems" ([Den76]). This principle was described under different names by many contemporary papers, including [SS75].

> "The computing environment [should be] designed so that no procedure has more capabilities than required for its immediate task."

There are many types of possible failure in a computer system, for example, a hardware failure, or an incorrect algorithm. If any type of failure occurs in a protection system, then the potential exists for access to be allowed in error. In this thesis, we focus only on one type of failure, a failure in the correctness of the design of the protection system. However, by this very design, with the principle of minimal privilege, we can also mitigate the consequences of such a failure. This principle does not prevent a failure from occurring, but it does limit the risk posed by any type of failure of the protection system.

2.4.2 The Principle of Attenuation of Privileges

The "Principle of Attenuation of Privileges" was studied in depth by Minsky ([Min78]), but also originates with Denning ([Den76]). Minsky's informal statement of the principle seems more concise.

> "Privileges should not be allowed to grow when they are transported from one place in the system to another."

The "transport" of rights is difficult to define precisely, although the intent is clear in most practical circumstances. Certainly, it includes the granting of rights: a principal should not be allowed to grant rights which it does not hold. While this principle appears obvious, there was at least one system contemporary with [Min78] which did not accept this principle, called Hydra ([WCC$^+$74], [CJ75]). In fact, the topic of [Min78] was to show that the Hydra system was vulnerable because it did not satisfy this principle.

2.4.3 Other Requirements

In the introduction to this thesis, we suggested many requirements which, while not necessarily functional or directly concerned with the operation of the protection system, are important if the protection system is to be of practical use. The satisfaction of these requirements is not core material of this thesis, but we would be gravely disillusioned were we to claim to construct a useful protection system without at least a nod to these requirements. In fact, in section 7 and section 10, we do considerable work towards satisfying these requirements; but these requirements remain optional, and are satisfied by very few current protection systems.

Low computational cost for the system: In the normal course of computation, it will rarely be the case that permission for an operation is denied: the denial is the exceptional case, rather than the matter of course. The protection system plays no part in the computation itself; it is primarily a safeguard against accidental or deliberate *unintended* operations. It is fair to require, therefore, that the protection system introduce minimal computational overhead to the system, since it contributes nothing to the objective of any computation. Any protection system which is correct but infeasibly expensive to operate is of no use, and we must therefore show that any proposed protection system will execute without significant overhead.

Administrative flexibility and convenience: As the system must have low computational overhead for the hardware, it must also have low administrative overhead for the users and administrators, otherwise again it fails to benefit those users. Current systems tend to be extremely poor in this regard.

Secure delegation of security system administration: The ability to delegate control is a sufficiently important element of administrative flexibility and convenience that it merits separate mention. It is for many reasons poor business practice for an organisation to rely on one person in any role, and this guideline naturally applies to the administration of a protection system. Just as it is impossible for every decision in a company to be made by one company chairman, it is impossible for all administration of a protection system to be done by a single administrator. Current protection systems provide very limited delegation.

Human-oriented justification of all access decisions: Increased delegation also increases the distribution of knowledge about the justification for particular decisions. As the system complexity increases, there may not even be human knowledge about why a particular permission exists. A simple audit trail may be too low level or contain insufficient information for justification of access decisions. More powerful systems must evolve which are capable of maintaining this information and representing it to the user.

Availability of accounting and auditing information: This is a popular requirement for computer security systems in critical applications. An accounting trail for such a system should contain enough information to permit the identification of unforeseen problems after the fact.[9]

[9] In other words, whenever the system administrator says, “Oops!”

2.5 Summary

A protection system is a self protecting mechanism which must prevent errors in any program or subsystem from adversely affecting any other program or subsystem with which it shares resources. Such errors might be:

- Hardware faults.
- Accidental software or other human errors.
- Generated with malicious intent.

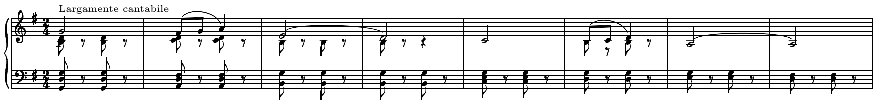
This mechanism is expressed in terms of rules for the manipulation of principals and permissions, and consists of:

1. An algorithm to decide the current principal.
2. An algorithm to decide whether a given principal may access a given object.
3. Optionally, a mechanism to decide whether a given principal may modify a given privilege assignment.

Material which follows from this section may be found in:

- Section 3: Introducing policy: the basis for our definitions of correctness, and justification for safety.
- Section 4: Introducing the formalism for protection systems and an analysis of safety.
- Section 8: How to compute with principals and correctly identify a current principal.

Where do protection systems come from . . . ?



3 Policy

A protection system can be an immensely complicated set of rules governing the use and modification of permissions. It is fair to wonder why these rules are constructed, and even, with a little foresight, whether there is a good definition of the '*correctness*' of a protection system. This definition is provided by '*policy*'. The study of policy is an immense field in itself, but we require only enough knowledge of policy to give an understanding of correctness in protection systems and motivate the safety problem.

3.1 An Introduction to Policy

A '*Policy*' is a set of pragmatic requirements, which may be expressed formally or informally, and form the basis for a definition of the correctness of a protection system: any protection system which satisfies all the requirements of a given policy may be adjudged correct with respect to that policy ([DoD85]). However, as the requirements for protection vary between organisations, no one policy gives universal definition of the correctness of a protection system. Any organisation requiring a protection system must construct a policy to describe its own specific requirements, and will then construct a protection system to satisfy this policy.

The idea of correctness of the protection system with respect to a policy is a driving theme of this thesis. We have traditionally compensated for our inability to show that a policy is satisfied by instead attempting to satisfy an overly restrictive[10] policy. The apparent security-flexibility trade-off is a side-effect of this tendency. When we have the ability to analyse a protection system and state whether it satisfies a policy, we may safely define extremely flexible policies which better suit our requirements; any policy we care to define is guaranteed to be enforced

[10] Paranoid, *adj*: Relating to or affected by paranoia: a strong, usually irrational, feeling [. . .], resulting in a tendency to be suspicious and distrustful, and to become increasingly isolated. ([Cha03])

by a protection system which is shown to satisfy it.

Policy is the vital missing link between the real world and the questions asked of the theoretical model in section 4. The general safety problem, as defined in section 4.4, is really the satisfiability problem for a policy statement. We must therefore introduce and gain a little understanding of the concept of policy in order that we may understand how to apply the later formalisms.

One way of tackling the formalisation of policy that has received much attention depends on various forms of logic (defeasible, deontic ([MW93], [Ort96]) and the use of legal reasoning ([Gor03], [Pra03]). We take a different direction because our priority is the construction of a protection system from a policy, and we would rather sidestep the hard problems inherent in the policy itself. In consequence, we make a considerable but not catastrophic restriction on the form of the statements in a policy before we may use it for practical analysis.

3.2 A Definition of Policy

The objective of a security policy is to specify which operations may be performed on the objects within a protection system or on the system itself. We formally define policy as follows.

Definition 3.1 (Security Policy). A security policy is a set of conditions which must hold for all reachable states of a protection system.

Definition 3.2 (Satisfaction). A protection system satisfies a policy if every reachable state of the system satisfies every predicate in the policy simultaneously.

The predicates of a policy may be expressed in natural language or in some formal notation. The only useful restriction we make immediately is that for a given state of a given protection system, it is possible to state whether each predicate in a policy holds.

In the United Kingdom, national and EU laws including especially those related Data Protection Act ([Cop98]), various evaluation frameworks ([DoD85], [Com99]), and guidelines such as BS7799 ([IOS00]) all suggest elements of security policy. However, there is no one-size-fits-all. A policy will usually be specific to the organisation for which it was constructed, and must encapsulate all the laws, rules, guidelines, and frequently the common practices of the organisation. There is much flexibility in policy design, and designing a policy is not easy. Luckily though, we are able to leave the design of policy to the lawmakers, and concentrate on the analysis and correct implementation of a given policy.

3.3 Making Use of Policy

The objective of a protection system is to answer *all* questions of access in a way that satisfies the security policy. A protection system is a completion of policy, in that it must specify a behaviour for every possible situation, whereas a policy need only specify behaviours for situations of interest to the policymaker. The protection system is thus responsible for the enactment of a policy. The policy is the law, and the protection system is the police: If a law is described in a policy, then a protection system which satisfies that policy will enforce that law, but may make an arbitrary decision in any case where policy is not applicable.

The objective of a systems analyst inspecting a protection system is to answer the question, "*Does this protection system satisfy the security policy?*" However, the ability to prove the correctness of the implementation with respect to the security policy is a tool which is not currently used in any formal methodology of protection systems. In fact, there is very little historical precedent for this formal use of policy. Harrison, Ruzzo and Ullman ([HRU76]), who introduced the concept of *safety*, left it implicit that the analyst would understand how to apply the analysis; but then, since most of the results from that era are negative (see section 6), little practical analysis was done, and thus a formal definition of policy was unnecessary.

3.3.1 A Minor Restriction of Policy

We cannot study policies containing arbitrarily complex requirements lest we detour into defeasible or deontic logic and hence lose sight of our objective: the use of policy in the analysis of protection systems. A policy might, for example, require that a protection system only allow access after some difficult undecidable problem has been solved.

For practical purposes, we will make a restriction on the form of the statements in a policy. The individual requirements set out by a policy *may* be entirely arbitrary, but in fact, the objectives of most system security policies are very similar.[11] We can express a wide range of common security policies with statements of only the following forms with very little loss of generality.

- A class of principals $[s]$ **must** be able to access a class of objects $[o]$.
- A class of principals $[s]$ **must not** be able to access a class of objects $[o]$.
- A class of principals $[s]$ **must** be able to modify the access specification of a class of objects $[o]$.
- A class of principals $[s]$ **must not** be able to modify the access specification of a class of objects $[o]$.

These wordings are only a guideline, and are far less precise than is usually found in a policy. They serve to describe only the idea of the restriction and have no formal basis. We can in fact be surprisingly flexible with the wording of our choice of statements without significantly altering the analysis, and it should also become clear how to extend our techniques to allow any more specific requirement which might arise in practice.

The statements in our security policy now fall into two categories: positive and negative. The positive statements are of the form, "*... must be able to ...*" and the negative statements are of the form, "*... must not be able to ...*". This is in contrast to our design choice for

[11] See section 2.1.1.

a protection system in section 2.3.1 where we stated that a protection system must use only positive tests in deciding access.

The statements of policy are also now entirely independent of one another. This is not in fact necessary; it would not change the analysis if requirements had an order of priority such that a more important requirement would override a less important requirement. Any more complex dependency between statements in policy is probably to be avoided, since then the policy itself would contain a logical satisfiability problem.

Under this restriction, all the statements in our policy are of fairly simple form, and we may check that the protection system satisfies each requirements independently. This *should* be a fairly simple task, yet it is not so. There might not even *exist* a protection system to satisfy a policy.

3.4 Some Considerations for Policy

Even without the problems of the more advanced logics, constructing a good policy is not easy and there are many traps and pitfalls for the unwary. We describe two such here: inconsistency and incompleteness. Of the two, inconsistency is immediately fatal, since there exists no protection system to satisfy an inconsistent policy. Incompleteness, on the other hand, is desirable if the resulting protection system is to have any flexibility, but a careless omission on the part of the policymaker may cause serious problems for the organisation.

3.4.1 Inconsistency of Policy

As with any set of requirements, a policy is not necessarily internally consistent and may contain contradictions. Policies are frequently derived from legislation, including that relating to copyright, data protection and privacy, and rules of organisational management. Such rules and laws are themselves not always consistent about what should be allowed and what should not be allowed, and may very easily lead to an inconsistent policy. This is a major problem, because it is not possible to construct a protection system to satisfy an inconsistent policy, and the "general consistency problem", to show whether an arbitrary given policy is consistent or satisfiable, is undecidable.

Consider the following policy:

1. Every accountant is an employee.
2. Every accountant must be able to modify every salary.
3. No employee must be able to modify his own salary.
4. There must exist a principal, Alice, who is an accountant.

By items (2) and (4), Alice can modify any salary in the company, necessarily including her own. However, this conflicts with rule (3), that Alice must not be allowed to modify her own salary. This is an example of an inconsistent policy, and it is impossible to construct a protection system which satisfies such a policy, because such a protection system must both allow and deny Alice access to modify her own salary.

While it may be possible to show that a particular policy is consistent, the general consistency problem for an arbitrary given policy is undecidable. We do not yet have the tools available to prove this, and later it will become less important, but a sketch proof is as follows:

- There exists a specific protection system for which the safety problem is undecidable.[12]
- A policy may specify a protection system completely.
- There exists a policy which specifies that an undecidable protection system is safe.
- Such a policy *may* be inconsistent; this is undecidable.

The reader is recommended not to give further consideration to this proof until the necessary material has been introduced, at which point it will become clear. This is the problem of inconsistency. It will be assumed in future sections that all policies are consistent.

3.4.2 Incompleteness of Policy

The order in which we specified the rules of our example policy is suggestive of a second possible issue. The policy given by rules (1), (2) and (3) of our example policy is entirely consistent, but does not admit the creation of any accountants. It is not until the configuration change introduced by rule (4) that the problem arises. A system can satisfy this policy while there are no accountants, but as soon as an accountant is introduced to the system, policy is no longer satisfied.

Complete information about all possible future operations and participants in any dynamic system is very rarely available. New objects and principals may enter a dynamic system at any time, but due to this incomplete nature of a policy, it may not be possible to deduce from the policy whether or not to permit an operation involving a new principal or object. However, the protection system is still required to make access decisions under these circumstances.

This issue is not as pivotal as that of inconsistency, since in this case there exists a good conservative strategy which may be followed by a protection system. In the discussion of axiom of positive tests in section 2.3.1, we stated that the default situation is for the protection system to deny access. What Glaser gave was an Asimovian "Zeroth Law of Policy", "*In all circumstances where this policy does not specify otherwise, deny access.*"

When this does not meet the needs of the organisation, the policy must still be considered to be at fault. Therefore, the systems designer has the problem of constructing a policy to satisfy future needs but with only partial information available about these future needs. This is the problem of incompleteness.

[12] See corollary 4.1 to come on page 44.

3.5 Satisfying a Policy

The objective of a system administrator in constructing a protection system is to create a system which accurately models, and thus satisfies the security policy. However, in the case that the policy requires dynamism in the protection system, this is not a simple task.

Consider that we must check whether a protection system satisfies a simple negative requirement: that a principal s not be able to access an object o. We must check not only whether the principal s may access object o, but whether the principal s may change the specification of who may access the object o, whether the principal s may change the specification of who may change the specification of who may access the object o and so forth recursively. In fact, given a dynamic protection system and an initial state for that system, we must check that the requirement holds in all states of the protection system reachable from the given initial state. Worse, we noted in section 2.3.4 that we do not necessarily assume the number of states of the system to be finite, so we cannot simply search reachable states. If we are to decide the correctness of a dynamic protection system, we must therefore analyse the mechanisms for the modification of access specifications mandated by section 2.1.4, to decide whether access specifications within the protection system may be modified to violate the given requirement.

We do have some points to our advantage in this analysis. First, axiom 2.3, the axiom of closure or self-protection guarantees that all the mechanisms for the modification of privileges exist entirely within the system, and that an analysis of these mechanisms will suffice, if it is possible. Second, the individual requirements of policy may be considered largely independently while they are of the forms discussed above. It is possible to verify that a state of a protection system instantaneously satisfies a given policy by verifying that it satisfies each statement of policy individually. If it satisfies every statement in the policy, then it satisfies the policy.

The problem now posed is very similar for both positive and negative statements, the question we must answer in either case is,

"*Can the privilege* (s, o, r) *be created through the normal operation of the system?*"

This is an informal statement of the *general safety problem*, introduced by [SS75] and extended considerably by [HRU76]. We will call an undesirable creation of a privilege a "*leakage*" of that privilege.

Given an algorithm to answer this question, the system administrator may then hypothesise a system in which certain modifications have been made to the assignment of permissions and execute the algorithm to test the safety of the system with respect to particular objects. We would therefore like to construct an algorithm test_security$(\Psi, Q, s, o, r, \ldots)$ which tests, for a protection system Ψ, a state of the system Q, a principal s, an object o and a right r, whether the privilege (s, o, r) may be created within the system. Unfortunately, as previously suggested, section 4.4.2 on page 41 proves this undecidable in general. This problem is the focus of the first part of this thesis.

3.6 Constructing a Protection System

If we cannot decide the correctness of an arbitrary protection system, perhaps we can explicitly construct a protection system to satisfy a given policy. By the nature of such a hypothetical construction, satisfaction of policy could be automatic. Unfortunately, this is also impossible: Consider a simple policy.

1. Alice must be able to modify the accounts.
2. Alice must be not able to modify the accounts.

Rule (1) is inconsistent with rule (2), and thus there exists no protection system to satisfy this policy. Worse still, a sketch proof in section 3.4.1 shows that deciding whether a policy is consistent is itself undecidable.

So what can we do? We can at best establish some guidelines for the construction of a protection system from a policy, such that it is *likely* to satisfy the policy, and such that it is *easy* to verify the system correct. This has historically been a popular approach. Common guidelines would include the following.

- **Make the system represent policy as closely as possible:** The less new material introduced by the protection system, the less the likelihood of introducing an inconsistency not present in policy, and therefore the better the chances of satisfying the policy.
- **Make the system minimally permissive:**[13] Since modifications may only be performed based on the presence of privileges, the less permissive the system, the less the likelihood that the system may be modified to violate policy.
- **Keep the system simple:** A simple system is likely to be easier to analyse and understand than a complex system and thus less likely to contain accidental mistakes.

Again, the balance between permissiveness and security rears its ugly head. A minimally permissive system is "more likely" to be secure. But in section 1.2 in the introduction to this thesis, we stated that a system is either secure, or it is not: this "middle ground" does not actually exist. It is merely an illusion created by our lack of knowledge.

Yet all is not lost! There is hope for future generations. There do exist some systems for which an analysis is possible. We describe some systems for which an analysis is possible in section 6, and when even those systems are shown to be unsatisfactory for a truly *practical* analysis, we construct yet more systems for which the analysis *is* practical in section 7.

[13] It is important to note the distinction between being minimally permissive, and the principle of minimal privilege from section 2.4.1. A system which is minimally permissive is one which conservatively prevents some operations. The principal of minimal privilege states that any principal must hold only the minimal privilege necessary to perform any permitted operation, but gives no guidance as to which operations are to be permitted or prevented.

3.7 Summary

We now have a clear objective for one of the main foci of this thesis: We must analyse the mechanism for the modification of the protection system (as described in section 2.1.4) to ensure that it satisfies a policy. We will produce a formalism as a basis for this analysis in section 4.

Policies are:

- Not necessarily complete.
- Not necessarily consistent.
- Allowed to contain arbitrarily complex statements.

Henceforth, we can safely ignore incompleteness, we assume consistency, and we focus on simple positive and negative statements.

We wish to evaluate whether any principal may gain access to an object in violation of the system security policy:

- By subverting an incorrect algorithm in the design of the protection system.
- Indirectly by modification of privilege assignments within the normal operation of the protection system.

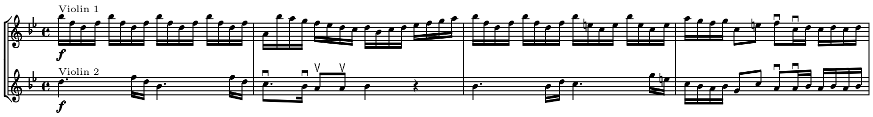
We must show that we can build systems which are:

- Correct: The algorithms may not be subverted.
- Safe: The algorithms may be analysed for accuracy of implementation of policy.

Material which follows from this section may be found in:

- Section 4: Introducing the formalism for protection systems and an analysis of safety.
- Section 7: The construction of protection systems for which analysis is practical.
- Section 12: Some proposals as to which classes of policy might be satisfiable by particular classes of protection system.

So how do we formalise this . . . ?



4 A Formal Model of Protection Systems

Section 2 introduced us to the workings of the world which we wish to formalise, and we should by now be fairly familiar with the concepts of protection systems. Our objective for this section is to produce a formalism of the concept of "*protection system*", the mechanisms of which were described very roughly in section 2.1.3. Any formalism we produce must be amenable to verification against a policy, as described in section 3.5.

4.1 Introduction to the Formalism

In 1976, another of the early seminal works in security appeared. In a work entitled, "Protection in Operating Systems" ([HRU76]), Harrison, Ruzzo and Ullman described a good generalised formalism for protection systems which was derived directly from the fairly simple systems available at the time. This formalism rapidly became the de facto standard in the literature surrounding protection systems, and was used in what limited formal analysis followed. Since no major positive results were achieved using this formalism, it appears to have fallen into disfavour, and is nowadays less often used.

We will commence the first of our main studies by reproducing in section 4.3 the formal generalised model of protection system found in [HRU76]. This is a generalisation of mechanisms found in current computing systems for controlling the modification of rights (as required by section 2.1.4).

Harrison, Ruzzo and Ullman realised, and described in [HRU76], the importance of being able to decide the safety of a particular protection system. Lacking our notion of policy, they stated instead that "*Basically, safety means that an unreliable [principal] cannot pass a right to someone who did not already have it.*" The decision process for this problem is called the "*general safety problem*", and it is the key to the verification of a protection system against a policy, as described informally in section 3.5. We will formally describe the general safety problem and reproduce the proof from [HRU76] that this problem is undecidable.

The formal model from [HRU76] contains many flaws and is incomplete in several respects, but introduces some critical concepts and proofs on which we will later build more complex and complete ideas. In section 4.6, we demonstrate a major failure of the model with respect to the basic mechanisms of section 2.1.3, and therefore we correct the model to include our concept of the *current principal.* This will extend [HRU76] to cover our full model of protection as described in section 2.1.3. This modification will be shown to have a surprisingly significant impact on the general safety problem.

4.2 Preliminary Definitions

Objects are still defined as in basic definition 2.6 on page 16. We will refer to the set of objects as O and use instances $o_0, o_1, \ldots$. The following formal definitions are new to this thesis.

Definition 4.1 (Subject). A *subject* is an object which is a principal.

Harrison's subjects are opaque, distinct objects of no innate computational use. Their nature as principals allows their use as labels for rows of an access matrix representation of privileges, as shown in figure 2 on page 36. Their nature as objects allows privileges to be held over them as if they were concrete processes (or similar) in the computer system.

We will refer to the set of subjects as S and use instances $s_0, s_1, \ldots$. We note that $S \subseteq O$. A subject may be referred to as an object when it is being acted upon and as a subject when it is the actor. This implies nothing about the nature of the subject and is intended only to clarify the role the subject is taking in the interaction.

Definition 4.2 (Right). A right is a token from some finite alphabet.

A right is the third element of a privilege (s, o, r), in the context of definition 2.8 on page 16. We will refer to the set of rights as R and use instances $r_0, r_1, \ldots$. Rights are called *generic rights* in [HRU76] to emphasise that a particular right r (e.g. '*write*') may be used to access many objects in (presumably) related ways. Each subject may have several different rights to any given object.

Definition 4.3 (Access Matrix). The *Access Matrix* is a matrix P with a row for each subject and a column for each object. The cell $P[s, o]$ contains the rights $\{r \mid (s, o, r) \text{ is a privilege}\}$.

Notation. The set of rights $P[s, o]$ held by s over o is frequently denoted $[s, o]$ where P is implicit. We then say $r \in [s, o]$ to denote that the privilege (s, o, r) is represented in the matrix. As suggested in section 2.3.1, we do not use the statement $r \notin [s, o]$.

Definition 4.4 (Configuration of a Protection System ([HRU76])). A configuration of a protection system is a triple (S, O, P), where S is the set of subjects, O is the set of current[14]

[14] This use of the word "current" by [HRU76] is casual and should not be confused with our formal use of the term for the "current principal".

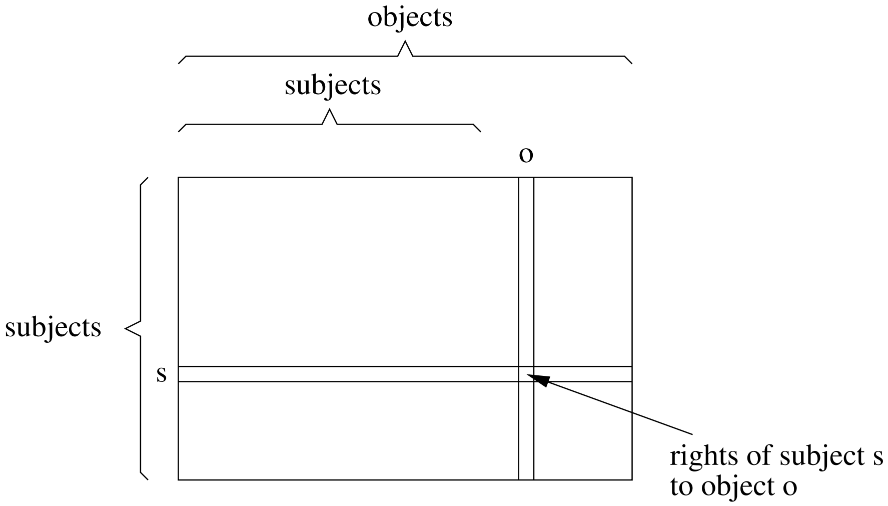


Figure 2: The access matrix

objects (with $S \subseteq O$) and P is an access matrix with a row for every subject in S and a row for every object in O. $P[s,o]$ is a subset of R, the generic rights. $P[s,o]$ gives the rights to object o possessed by subject s. The matrix is illustrated in figure 2.

4.3 A Formal Model of Protection Systems

We now reproduce the formal definition of protection system from [HRU76].

Definition 4.5 (Protection System ([HRU76])).
A protection system consists of:

1. A finite set of rights R.
2. A finite set C of commands of the form:

```
command $\alpha(X_1, X_2, X_3, \ldots, X_k)$
    if
        $r_0 \in [X_{s_0}, X_{o_0}]$ and
        $r_1 \in [X_{s_1}, X_{o_1}]$ and
        $\ldots$
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        op$_1$
        op$_2$
        $\ldots$
        op$_n$
end
```

where α is a name, $X_1, \ldots, X_k$ are formal parameters, X_{s_i} is one of the formal parameters X_j which is a subject, X_{o_i} is one of the formal parameters, and each op_i is one of the primitive operations

enter r **into** $[s, o]$
delete r **from** $[s, o]$
create subject s
destroy subject s
create object o
destroy object o

Our notation, given here, considerably augments that of [HRU76] in order to allow our more complex computations with protection systems in section 5.

Notation.

A protection system is denoted as Ψ where $\Psi = (R, C)$.

The size of a protection system Ψ is denoted $|\Psi|$ where $|\Psi| = |R| + |C|$.

A configuration of a protection system is denoted as Q where $Q = (S, O, P)$. It is implicit that a configuration is associated with the protection system of which it is a configuration.

The size of a configuration of a protection system is denoted $|Q|$ where $|Q| = |S||O|$.

The set of all possible configurations of a protection system is denoted as $\mathrm{cf}(\Psi)$, so $Q \in \mathrm{cf}(\Psi)$.

The definitions of the primitive operations are implied by their names. Formally, we may describe their effect on the configuration of a protection system.

Definition 4.6 (The Six Primitive Operations ([HRU76])).

$(S, O, P) \Rightarrow_{op} (S', O', P')$ if either

1. op = **enter** r **into** $[s, o]$, $S' = S$, $O' = O$, $s \in S$, $o \in O$, $P'[s', o'] = P[s', o']$ if $(s', o') \neq (s, o)$, $P'[s, o] = P[s, o] \cup \{r\}$.

2. op = **delete** r **from** $[s, o]$, $S' = S$, $O' = O$, $s \in S$, $o \in O$, $P'[s', o'] = P[s', o']$ if $(s', o') \neq (s, o)$, $P'[s, o] = P[s, o] - \{r\}$.

3. op = **create subject** s' where s' is a new symbol not in O, $S' = S \cup \{s'\}$, $O' = O \cup \{s'\}$, $P'[s, o] = P[s, o]$ for all $(s, o) \in S \times O$, $P'[s', o] = \emptyset$ for all $o \in O$, $P'[s, s'] = \emptyset$ for all $s \in S$.

4. op = **create object** o' where o' is a new symbol not in O, $S' = S$, $O' = O \cup \{o'\}$, $P'[s, o] = P[s, o]$ for all $(s, o) \in S \times O$, $P'[s, o'] = \emptyset$ for all $s \in S$.

5. op = **destroy subject** s' where $s' \in S$, $S' = S - \{s'\}$, $O' = O - \{s'\}$, $P'[s, o] = P[s, o]$ for all $(s, o) \in S' \times O'$.

6. op = **destroy object** o' where $o' \in O - S$, $S' = S$, $O' = O - \{o'\}$, $P'[s, o] = P[s, o]$ for all $(s, o) \in S' \times O'$.

This definition from [HRU76] is perhaps a little concise; a more verbose explanation of **enter** r **into** $[s, o]$ might help fix the ideas in the mind of the reader.

$S' = S$	The set of subjects is unchanged.
$O' = O$	The set of objects is unchanged.
$s \in S, o \in O$	The specified subject and object exist.
$P'[s', o'] = P[s', o']$ if $(s', o') \neq (s, o)$	The contents of cells other than $P[s, o]$ is unchanged.
$P'[s, o] = P[s, o] \cup \{r\}$	r is added to the contents of the cell $P[s, o]$.

Definition 4.7 (Execution of the System ([HRU76])).
Let $Q = (S, O, P)$ be a configuration of a protection system (R, C) where C contains

```
command $\alpha(X_1, X_2, X_3, \ldots, X_k)$
    if
        $r_0 \in [X_{s_0}, X_{o_0}]$ and
        $\ldots$
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        $\mathrm{op}_1$
        $\ldots$
        $\mathrm{op}_n$
end
```

We say $Q \vdash_{\alpha(x_1, x_2, \ldots, x_k)} Q'$ where Q' is defined as follows:

1. If the conditions of α are not satisfied, i.e. if there is some $1 \leq i \leq m$ such that $r_i \notin [X_{s_i}, X_{o_i}]$, then $Q' = Q$.

2. Otherwise, for all i between 1 and m, $r_i \in [X_{s_i}, X_{o_i}]$, then let there exist configurations $Q_0, Q_1, \ldots, Q_n$ such that

$$Q = Q_0 \Rightarrow_{\mathrm{op}_1^*} Q_1 \Rightarrow_{\mathrm{op}_2^*} \ldots \Rightarrow_{\mathrm{op}_n^*} Q_n$$

   where op_i^* denotes the primitive operation op_i with the actual parameters $x_1, \ldots, x_k$ replacing all occurrences of the formal parameters $X_1, \ldots, X_k$ respectively. Then $Q' = Q_n$.

We say that $Q \vdash_\alpha Q'$ if there exist parameters $x_1, \ldots, x_k$ such that $Q \vdash_{\alpha(x_1, x_2, \ldots, x_k)} Q'$.

We say that $Q \vdash Q'$ if there exists $\alpha \in C$ such that $Q \vdash_\alpha Q'$.

We also write $Q \vdash^* Q'$ where $\vdash^*$ is the reflexive and transitive closure of $\vdash$, that is, $\vdash^*$ represents zero or more applications of $\vdash$.

The access matrix commands are the only way to modify the access matrix. The matrix itself is protected from arbitrary interference, as described in section 2.3.4. The matrix, together with the access matrix commands, clearly matches the definition in section 2.1 of a protected object. However, the commands are considerably more complex than the simple tests for the existence of a right ($r \in [s, o]$) to which we are more accustomed.

This concludes our reproduction of the formalism from [HRU76].

4.4 The General Safety Problem

In order to demonstrate that a particular rule of a security policy is satisfied by a protection system, we need to be able to answer the question posed by the general safety problem, as introduced briefly in section 3.5. However, in the case of even a simple requirement, it is impossible to decide whether the requirement is satisfied in all reachable configurations of the protection system. We will now reproduce the proof from [HRU76] to show that the general safety problem for the formal model of protection systems is undecidable.

Definition 4.8 (The General Safety Problem ([HRU76])). Given a protection system and an initial configuration for that system, can a subject s acquire the right r to access an object o?

This question is sometimes phrased, "*Can s r o?*" as if the right r were a verb, implicitly "*Can s perform the operation for which right r is required on object o?*". Can s acquire, directly or indirectly, the key r to the box containing object o through the normal mechanisms of the given protection system? In section 3.5, this question was phrased, "Can (s, o, r) be created through the normal operation of the system?" A means of answering this question is essential for the verification of policy as described in section 3.5.

4.4.1 An Example of the Safety Problem

Example 4.1 (A Simplified Take-Grant Model (Simplified from [JLS76])). We hypothesise a system with the set of rights $R = \{r, w\}$ where r represents read and w write permission on the privileges of a target subject. The commands of the system are given in figure 3 on the following page. This example is a simplification of a model presented in [JLS76] and closely models the design of some capability systems ([SSF99], [NH82], [NW74]).

Figure 4 demonstrates a scenario in which a subject S has write permission over a subject T and the subject T has an unspecified permission a over the object O. The question we pose as an example of the safety problem for this protection system is "*Can S acquire right a over object O?*" The answer is *yes*; we describe each step in figure 4 as follows.

1. The initial configuration.

2. S creates a new subject X, over which it has read and write permissions.

3. S grants read and write permissions over X to T.

```
command create($S, O$)
    create subject $O$
    enter $r$ into $[S, O]$
    enter $w$ into $[S, O]$
end
```

Allow the creation of a new subject over which we hold all privileges.

```
command take$_x$($S, O, P$)
    if
        $r \in [S, O]$ and
        $x \in [O, P]$
    then
        enter $x$ into $[S, P]$
end
```

Allow us to take the privilege x from any subject over which we have the read permission r (for each x in $\{r, w\}$).

```
command grant$_x$($S, O, P$)
    if
        $w \in [S, O]$ and
        $x \in [S, P]$
    then
        enter $x$ into $[O, P]$
end
```

Allow us to grant the privilege x to any subject over which we have the write permission w (for each x in $\{r, w\}$).

Figure 3: Commands in the Take-Grant system

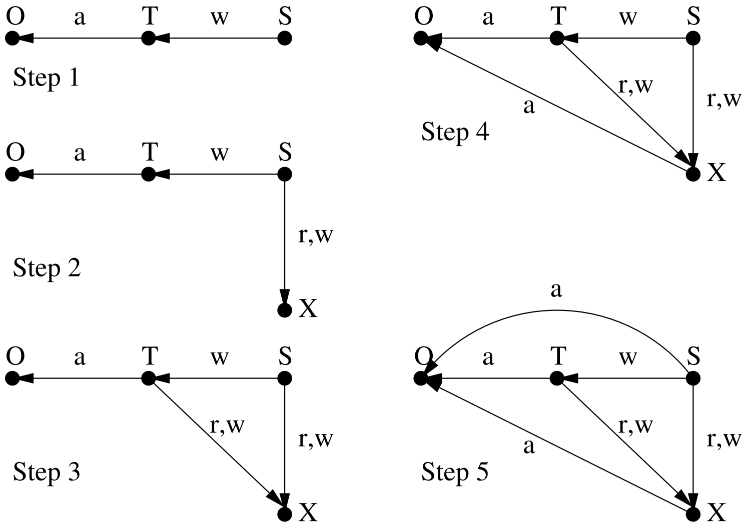
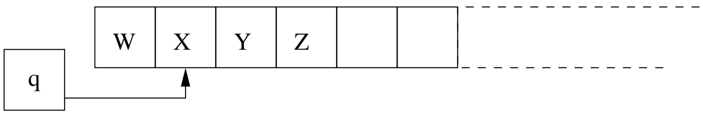


Figure 4: The leakage of the right a: a worked example

4. T grants a permission over O to X.

5. S takes a permission over O from X. Thus S indirectly acquires the permission a over O



	s_1	s_2	s_3	s_4
s_1	$\{W\}$	$\{\mathbf{own}\}$		
s_2		$\{X, q\}$	$\{\mathbf{own}\}$	
s_3			$\{Y\}$	$\{\mathbf{own}\}$
s_4				$\{Z, \mathbf{end}\}$

Figure 5: Representing a Turing machine

with the collaboration of T.

4.4.2 Undecidability of the General Safety Problem

The general safety problem is undecidable. [HRU76] proved this result by showing that a protection system can emulate a Turing machine on the long diagonal of the access matrix. This reduces the general safety problem to the halting problem, which is known to be undecidable by the proof in appendix A.5. We will reproduce here the proof of the undecidability of the general safety problem from [HRU76].

Definition 4.9 (Turing Machine). A Turing machine T consists of a finite set of *states* K and a finite set of distinct *tape symbols* Γ. One of the symbols is the *blank* B, which initially appears on each cell of a tape which is infinite to the right only (that is, the tape cells are numbered 1, 2, ..., i, ...). There is a *tape head* which is always *scanning* (located at) some cell of the tape.

The *moves* of T are specified by a function $\delta : K \times \Gamma \rightarrow K \times \Gamma \times \{L, R\}$. If $\delta(q, X) = (p, Y, R)$ for states p and q and tape symbols X and Y, then should the Turing machine T find itself in state q, with its tape head scanning a cell holding symbol X, then T enters state p, erases X and prints Y on the tape cell scanned and moves its tape head one cell to the right. If $\delta(q, X) = (p, Y, L)$, the same thing happens, but the tape head moves one cell left (but never off the left end of the tape at cell 1).

Initially, T is in state q_0, the *initial state*, with its head at cell 1. Each tape cell holds the blank. There is a particular state q_f, known as the *final state*, and it is a fact that it is undecidable whether started as above, an arbitrary Turing machine T will ever enter state q_f (see appendix A.5).

Theorem 4.1 (Undecidability of the General Safety Problem ([HRU76])). *It is undecidable whether a given configuration of a given protection system is safe for a given right.*

Proof. We shall show that the general safety problem is undecidable by showing that a protection system can simulate the behaviour of an arbitrary Turing machine, with leakage of a right corresponding to the Turing machine entering a final state, a condition we know to be undecidable. The set of generic rights of our protection system will include the states and tape symbols of the Turing machine.

At any time, the Turing machine will have scanned some finite initial prefix of its tape cells, say $1, 2, \ldots, k_i$. This situation will be represented by a sequence of k subjects, $s_1, s_2, \ldots, s_k$, such that s_i "*owns*" s_{i+1} for $1 \leq i \leq k$. This relationship orders subjects into a linear linked list representing the tape of the Turing machine. Subject s_i represents cell i, and the fact that cell i now holds tape symbol X is represented by giving s_i generic right X to itself. The fact that q is the current state and that the tape head is scanning the j^{th} cell is represented by giving s_j generic right q to itself. Note that we have assumed the states distinct from the tape symbols, so no confusion can result.

There is a special generic right **end**, which marks the last subject s_k. That is, s_k has generic right **end** to itself, indicating that we have not yet created the subject s_{k+1} which s_k is to own. The generic right **own** completes the set of generic rights. An example showing how a tape whose first four cells hold *WXYZ*, with the tape head at the second cell and the machine in state q, is shown in figure 5.

The moves of the Turing machine are reflected in commands as follows. First, if $\delta(q, X) = (p, Y, L)$ then there is a command:

```
command $\mathrm{C_{qX}}(s, s')$
    if
        own $\in [s, s']$ and
        $q \in [s', s']$ and
        $X \in [s', s']$
    then
        delete $q$ from $[s', s']$
        delete $X$ from $[s', s']$
        enter $p$ into $[s, s]$
        enter $Y$ into $[s', s']$
end
```

That is, s and s' represent two consecutive cells of the tape, with the machine in state q, scanning the cell represented by s', and with the symbol X written in s'. The body of the command changes X to Y and moves the head left, changing the state from q to p. For example, figure 5 becomes figure 6 when command C_{qX} is applied.

	s_1	s_2	s_3	s_4
s_1	$\{\mathbf{W}, \mathbf{p}\}$	$\{\textbf{own}\}$		
s_2		$\{Y\}$	$\{\textbf{own}\}$	
s_3			$\{Y\}$	$\{\textbf{own}\}$
s_4				$\{Z, \textbf{end}\}$

Figure 6: After one move

If $\delta(q, X) = (p, Y, R)$ that is, the tape head moves right, then we have two commands, depending on whether or not the head passes the current end of the tape, that is, the **end** right. There is a command:

```
command $\mathrm{C_{qX}}(s, s')$
    if
        own $\in [s, s']$ and
        $q \in [s, s]$ and
        $X \in [s, s]$
    then
        delete $q$ from $[s, s]$
        delete $X$ from $[s, s]$
        enter $p$ into $[s', s']$
        enter $Y$ into $[s, s]$
end
```

To handle the case where the Turing machine moves into new territory, there is also a command:

```
command $\mathrm{D_{qX}}(s, s')$
    if
        end $\in [s, s]$ and
        $q \in [s, s]$ and
        $X \in [s, s]$ and
    then
        create subject $s'$
        enter own into $[s, s']$
        delete end from $[s, s]$
        enter end into $[s', s']$
        enter $B$ into $[s', s']$

        delete $q$ from $[s, s]$
        delete $X$ from $[s, s]$
        enter $p$ into $[s', s']$
        enter $Y$ into $[s, s]$
end
```

If we begin with the initial matrix having one subject s_1 with rights q_0, B (blank) and **end** to itself, then the access matrix will always have exactly one generic right that is a state. This follows because each command deletes a state known by the conditions of that command to exist. Each command also enters one state into the matrix. Also, no entry in the access matrix can have more than one generic right that is a tape symbol by a similar argument. Likewise, **end** appears in only one entry of the matrix, the diagonal entry for the last created subject.

Thus, in each configuration of the protection system, reachable from the initial configuration, there is at most one command applicable. This follows from the fact that the Turing machine has at most one applicable move in any situation, and the fact that C_{qX} and D_{qX} can never be simultaneously applicable. The protection system must therefore exactly simulate the Turing machine using the representation we have described. If the Turing machine enters state q_f then the protection system can leak generic right q_f, otherwise it is safe for q_f. Since it is undecidable whether the Turing machine enters state q_f, it must be undecidable whether the protection system is safe for q_f.

□

Here ends our reproduction of the proof of the undecidability of the general safety problem. [HRU76] also produces three immediate corollaries of this proof.

Corollary 4.1. *There exists a particular protection system for which there is no specific algorithm to decide safety.*

Proof. By simulating a universal Turing machine on an arbitrary input, we can exhibit a particular protection system for which it is undecidable whether a given initial configuration is safe for a given right.

Thus, although we can give different algorithms to decide safety for different classes of system, we can never hope to cover all systems with a finite, or even infinite, collection of algorithms.

□

Corollary 4.2. *Since arbitrarily complex computable functions exist, there exist protection systems for which safety is decidable but arbitrarily difficult.*

Corollary 4.3. *The question of safety for protection systems without create commands is complete on polynomial space.*

Proof. A construction similar to that of theorem 4.1 proves that any polynomial space bounded Turing machine can be reduced in polynomial time to an initial access matrix whose size is polynomial in the length of the Turing machine input.

□

4.5 Consequences of the Definition

We introduce some notes on the formal definition of a protection system to aid with later explanations.

4.5.1 Graph Representations of Configurations

A graph representation of configurations was used in [JLS76] without obvious precedent, although the idea is obvious and was probably discussed without record at the time.

The matrix

	a	b	c	d	e
a		r,w			
b			r	w	
c					r
d					r, w

The equivalent graph

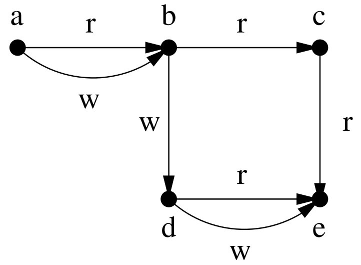


Figure 7: Matrix and graph representations of a configuration

We may consider the access matrix to be the adjacency matrix for a graph in which all edges are directional and tagged with some subset $A \subseteq R$. In this case, the existence of the privilege (s, o, r) would be represented by an edge $s \xrightarrow{r} o$.[15]

A configuration $Q = (S, O, P)$ of a protection system Ψ may be transformed into an equivalent graph $G = (S, O, E)$ where O is the set of objects from the access matrix and now also the set of nodes of the graph, $S \subset O$ is also preserved, and E is the set of directed edges indicated by the presence of rights in P. The edges in E are discriminated by rights from R so that the edge $s \xrightarrow{r} o$ is in E if and only if $r \in P[s, o]$. Only those nodes representing objects which are also subjects in the original access matrix will have outgoing edges. An example of a graph representation of an access matrix may be seen in figure 7 on page 45.

Access matrix commands in C test for the existence of a particular subset of rights in the access matrix before modifying the matrix. Equivalently, in the graph representation, a command will test for the existence of a particular (not necessarily connected) subgraph in the guard clause, and make modifications to the graph if this structure is found.[16]

4.5.2 The Difficulty of Search

Deciding the general safety problem would involve either an abstract analysis of the protection system, or a search of the space of reachable configurations of the protection system. The impossibility of abstract analysis may be hard to grasp without some thought, but it is perhaps easier to visualise the difficulty, if not necessarily the impossibility, of a search.

Consider a graph representation of a configuration, as proposed in section 4.5.1. It is possible to create an access matrix command which tests for the existence of a k-clique for arbitrary k. If even the decision problem for the applicability of even this one command is NP Hard, then the search problem for the space of the set of configurations is impractical even in simple cases.

[15] Occasionally we will perpetrate some abuse of notation (when ambiguity is not possible) by writing a set of rights as a label on a single arrow as $s \xrightarrow{A} o$ where $A \subseteq R$.

[16] It is not quite correct to say that a command tests for the existence of a particular subgraph: arguments passed to an access matrix command are not necessarily distinct. In fact it tests for the existence of a morphism from some structure defined by the guard clause of the command to some subgraph of the configuration.

If $P[s, s] = R$ for any s, then any command may be executed with s as every argument. A morphism exists from any graph mapping every node to s. This makes the selection of an appropriate structure in the proof of theorem 6.8 a little trickier than it might seem.

4.5.3 The Logical Power of Access Matrix Commands

By the distributive law, we may express any formula in boolean logic in disjunctive normal form, as a disjunction of conjunctions, $\bigvee \bigwedge \mathrm{atom}_{ij}$. Since the guard clause of an access matrix command is a conjunction, and the choice of commands allows a disjunction, the access matrix formalisation actually allows a protection system to express any formula in classical logic without negation: that logic using the connectives $\wedge$, $\vee$ and the atoms '$r \in [X_s, X_o]$'.

In practice and in discussion, a protection system may contain guard expressions using the boolean connective $\vee$ on the understanding that the distributive law allows these guard expressions to be re-expressed in Harrison's original model.

The restriction to positive atoms removes from the system the ability to directly express guard conditions such as 'debt $\Rightarrow$ payment' or "*If you have debts and have paid them or you have no debts*". However, under many circumstances we may allow the creation of a complement relation to the relation which we wish to negate. This can sometimes allow us to achieve an effective negation, as shown in example 4.4 on page 51.

4.5.4 The Simplicity of Object Protection Code

Our first informal example in section 2.1.2 required that a principal use only one key to access a protected object, as if the following code were present as a pseudo-command in every protection system.

Example 4.2 (Object Protection Code).

```
command access($O$)
    let $s$ = current_principal
    if
        $r \in [s, O]$
    then
        allow_access($O, r$)
    else
        fail
end
```

The alternative is to allow that a principal be required to hold n particular keys to access a protected object. This would make the study of safety considerably more complex since we must now test for the safety of many rights simultaneously. However, we may restrict our consideration to the safety of only the one key without significant loss of generality: Were a protected subsystem to require n rights $r_0, r_1, \ldots$ to access an object o, we may modify the system such that o may be accessed using some new right r_ν and add a command to grant r_ν to anybody holding the rights $r_0, r_1, \ldots$ to o.

```
**command** grant_$r_\nu(S, O)$
    **if**
        $r_0 \in [S, O]$
        $r_1 \in [S, O]$
        ...
    **then**
        **enter** $r_\nu$ **into** $[S, O]$
**end**
```

Thus it is sufficient and without loss of generality in the study of the safety problem that we consider only the safety of a single right. It is also unnecessary for our study to explicitly consider the mechanism used for accessing objects: we observed under section 4.5.3 that a set of access matrix commands can express any formula in classical logic without negation, thus we may reduce even the most complex case of object protection code to a test for a single right.

4.5.5 Countability of Configurations

Theorem 4.2 (Countability of Configurations). *If either of $|S|$ or $|O|$ is unbounded, then the number of configurations of the protection system is uncountable.*

Proof. A configuration of the protection system is an element of $2^{S \times O \times R}$. If either of S or O is infinite, then $S \times O \times R$ is infinite, and the number of configurations is uncountable by theorem A.9

□

4.6 The Introduction of the Current Principal

In any practical system, we usually require that we only ask for commands to be executed based on the privileges of the principal responsible for invoking a command: the current principal; not those of any number of arbitrary third parties.

Within the access matrix formalisation, the only principals are the subjects of the access matrix, and we may equally refer to the "*current subject*" to mean the current principal. But the current subject is conspicuous only by its absence from the formalism in [HRU76]. The formalism not only contains no concept of responsibility, but even given that we identify a responsible principal, this principal is not discriminated in any way in an access matrix command.

It is not immediately obvious that this omission is directly relevant to the application of the safety problem. In order to demonstrate the applicability, we will compare the application of his formulation of the safety question with a new formulation involving a current principal.

In order to make a practical application of an algorithm to decide the safety problem for a right r we must exclude from tests by our algorithm those subjects who may legitimately grant r but are trusted not to do so. [HRU76] proposes that:

> "It might also make sense to delete from the matrix any other 'reliable' subjects who could grant r, but whom s 'trusts' will not do so. It is only by using the hypothetical safety test in this manner, with 'reliable' subjects deleted, that the ability to test whether a right can be leaked has a useful meaning in terms of whether it is safe to grant a right to a subject."

However, the principle of trust is that a principal will not perform the action of granting the right r, not that the principal will not allow its rights to be exploited (if the access matrix commands allow this) to grant r. Indeed, a principal being exploited frequently has no control over whether its rights may be used in this manner. The only assertion satisfying the intuition is that the principal will not intentionally perform (and hence take responsibility for) an action leading to an exploitation of its rights.

What is missing entirely from Harrison's access matrix commands is this principle of the "*current principal*", or in this case, "*current subject*". When our analysis of the basic mechanisms of protection in section 2.1.3 requires us to identify the current principal in order to know which keys to use, why does the formulation of an access matrix modification command not require the same? This is in violation of reason and intuition.

We therefore propose a reformulation of the generic access matrix command in the form:

```
command $\alpha(X_1, X_2, X_3, \ldots, X_k)$
    let $X_0 =$ current_subject
    if
        $r_0 \in [X_{s_0}, X_{o_0}]$ and
        $\ldots$
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        $\mathrm{op}_1$
        $\ldots$
        $\mathrm{op}_n$
end
```

with all terms defined as before and the **current_subject** being defined as the current principal in definition 2.11 on page 17 and evaluated using the normal mechanisms of the protection system as required in section 2.1.3.

We will demonstrate the effects of this conceptual modification on the decision algorithm for the safety problem. The example we will use is a transitivity relation, common in similar forms in role based access control models and capability systems ([SCFY96], [SSF99]). An early formalisation of a similar rule was constructed by Lipton and Snyder in [JLS76]. We produce an example of a real world mechanism to which this example is pertinent in section 7.9.5.

Example 4.3 (Two Formulations of the Safety Problem). The transitivity relation may be expressed as follows.

The initial configuration

	s_0	s_1	o
s_0		r	?
s_1			r

The [HRU76] access matrix command

```
command transfer($S, T, O$)
    if
        $r \in [S, T]$ and
        $r \in [T, O]$
    then
        enter $r$ into $[S, O]$
end
```

After removal of the trusted s_1 in the formulation of the safety problem without a current subject, we get

	s_0	s_1	o
s_0		r	

The system becomes non-leaky since the *transfer* command cannot be executed.

The command with a current subject

```
command transfer($T, O$)
    let $s$ = current_subject
    if
        $r \in [s, T]$ and
        $r \in [T, O]$
    then
        enter $r$ into $[s, O]$
end
```

In the formulation of the safety problem with a current subject, no modifications are made to the matrix, and the command can be applied to the current matrix with s_0 as the current_subject. We execute transfer(s_1, o) to get

	s_0	s_1	o
s_0		r	**r**
s_1			r

The system is leaky (as expected).

Figure 8: Two formulations of the safety problem

$$\begin{array}{rc c c}
\textbf{If} & s_0 \text{ can access } s_1 & & r \in [s_0, s_1] \\
\textbf{and} & s_1 \text{ can access } o & \textit{equivalently} & r \in [s_1, o] \\
\hline
\textbf{then} & s_0 \text{ should be allowed to access } o & & r \in [s_0, o]
\end{array}$$

We will demonstrate the effects of the application of a command expressing this relation on a simple access matrix. Figure 8 displays two variants of the command; first without, and then with the current subject. An example matrix is given as an initial configuration for an instance of the safety problem: to decide whether the privilege $r \in [s_0, o]$ is safe. In the first case, the system is safe, and in the second case, it becomes leaky.

In this example, the reformulation of the access matrix command to include the current subject is shown to have a significant effect on the safety problem. In accordance with our introductory analysis in section 2.1.3, we must clearly prefer the formulation which includes the current subject, and thus we will correct the access matrix formalisation of protection systems to include the current subject in all access matrix commands.

In any algorithm to decide the safety problem, it will henceforth be assumed that any subject which might legitimately act as the current subject to grant the right $r \in [s, o]$ but is trusted not to do so does not do so. It is then assumed that all other users may collude to break the security of the system. Therefore, for any command with given parameters, we may choose an arbitrary *current subject* from the remaining untrusted subjects in the system, since whichever of the remaining subjects we might choose might be a member of the conspiracy.

4.7 Example: An Inverse Relation

Most policies contain restrictions of exclusion as well as restrictions of inclusion. Such a restriction might state that principal must not be permitted to access an object. But our protection system formalism does not allow for these conditions to be included in the guard of an access matrix command. We may ask $r \in [s, o]$ but not $r \notin [s, o]$. One might therefore consider the axiom of positive tests (axiom 2.1 on page 18) to be over-restrictive. However, the access matrix framework and the formulation of the safety problem are surprisingly powerful, and we can show that it is possible to construct systems which allow us to perform these apparently negative tests by constructing an inverse map and using only positive tests and the safety problem. We will demonstrate this technique by constructing an inequality relation as an inverse to an equality relation.

4.7.1 An Introduction to the Inverse Relation

Any subject is equal only to itself. We may construct an *always defined* equality relation within any protection system using no more access matrix commands than were previously required by entering $r_{\text{equal}} \in [s, s]$ immediately on creating any subject s. The term *always defined* means that either the right is present in the matrix, or the system is safe for the right. An algorithm to decide safety of such a right is always trivial; if the right is not present, it may never be present, although showing a right to be *always defined* may be difficult.

The r_{notequal} relation which we will attempt to construct is the inverse of this trivial equality relation. The new relation is not *always defined* in the previous sense; we must fall back to the safety problem to decide inequality by asking not, "*Is the right r_{notequal} present?*" but "*Is the system safe for the right r_{notequal}?*"

4.7.2 The Construction of an Inverse Relation

It is undesirable that a nuclear weapon may be triggered by any one person acting alone. It may be desirable to require that at least k people acting in concert must exercise their rights in order to trigger the weapon.

Without loss of generality, we will limit ourselves to the case $k = 2$ and require the collaboration of only two subjects to explode the bomb. Let B represent a bomb and S_0 and S_1 represent the two actors wishing to explode it. In order to represent the concept of joint action, we will temporarily introduce two current_subjects. We would like to use the following access matrix command to explode the bomb.

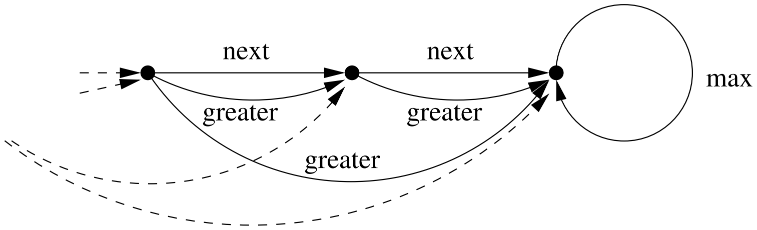


Figure 9: The construction of the relation r_{greater}

```
command explode($B$)
    let $s_0$ = current_subject_0
    let $s_1$ = current_subject_1
    if
        $r_{\text{explode}} \in [s_0, B]$
        $r_{\text{explode}} \in [s_1, B]$
        $r_{\text{notequal}} \in [s_0, s_1]$
    then
        enter $r_{\text{bang}}$ into $[B, B]$
end
```

Every subject should be related by the r_{notequal} relation to every other subject in the matrix; but the matrix contains a potentially unbounded number of subjects. The ability to grant these rights directly on creation of a new subject would require an unbounded number of access matrix commands, one for each possible number of subjects in the access matrix. Therefore we must grant the right by subterfuge.

We will construct a system such that it is possible to show that two subjects are distinct by induction using access matrix commands which eventually form a full strict order relation r_{greater} between subjects.[17]We can then construct an inequality relation using the axioms of an ordered set from the definition of an ordered set in definition A.1 on page 216. By this definition, two subjects are distinct if either one is '*greater than*' the other in the full order relation. Therefore if $r_{\text{greater}} \in [s_0, s_1]$ or $r_{\text{greater}} \in [s_1, s_0]$ then $s_0 \neq s_1$, and we may grant $r_{\text{notequal}} \in [s_0, s_1]$ and $r_{\text{notequal}} \in [s_1, s_0]$

Recall also that ordering is transitive. Therefore, if $r_{\text{greater}} \in [s_0, s_1]$ and $r_{\text{greater}} \in [s_1, s_2]$, then we may grant $r_{\text{greater}} \in [s_0, s_2]$ We will use this transitivity relation to build an unbounded amount of data from a finite number of access matrix commands. Now we can begin our example.

Example 4.4 (The Inequality Map). Without much loss of generality, we will ignore the case of the empty matrix, and assume that at least one subject exists. The subjects in the matrix are ordered by the relation r_{greater}, similarly to the organisation of the subjects representing the tape in section 4.4. The maximal subject of this order has the right r_{max} to itself. A fragment

[17] Unfortunately, we must introduce the assumption of countability on S, but we can lose it again in section 10.

of such an ordered set of subjects is illustrated in figure 9.

When we add a subject to the matrix, we will make it greater than the maximal element of the matrix, thus making it the new maximal element of the matrix. This new subject must then be tagged as the maximal element of the set of subjects.

```
command addsubject($E, S$)
    if
        $r_{\text{max}} \in [E, E]$
    then
        create subject $S$
        delete $r_{\text{max}}$ from $[E, E]$
        enter $r_{\text{max}}$ into $[S, S]$
        enter $r_{\text{next}}$ into $[E, S]$
        enter $r_{\text{greater}}$ into $[E, S]$
end
```

Now S is greater than E, the previous maximum and is itself tagged as the maximal element of the set of subjects. Now we exploit the transitivity of full ordering in order to build the complete ordering relation between all subjects. The right r_{next} is merely for the convenience of the later algorithm.

```
command order($S, T, U$)
    if
        $r_{\text{greater}} \in [S, T]$ and
        $r_{\text{greater}} \in [T, U]$
    then
        enter $r_{\text{greater}}$ into $[S, U]$
end
```

We must also be able to convert our strict ordering relation into an inequality.

```
command unequal($S, T$)
    if
        $r_{\text{greater}} \in [S, T]$
    then
        enter $r_{\text{notequal}}$ into $[S, T]$
        enter $r_{\text{notequal}}$ into $[T, S]$
end
```

Thus we have constructed an inequality relation.

As an example of interest, we will show that the protection system described in example 4.4 is safe for $r_{\text{notequal}} \in [s, s]$. This is not strictly necessary, since our construction of the system has ensured this property. However, the more general technique we will use to reprove this suggests a calculus of safety which we describe further in section 5.4.

Example 4.5 (A Specific Application of the Safety Problem). We will show the system safe for $r_{\text{notequal}} \in [s', s']$ for any subject s', starting from an initial configuration containing only one principal s and the privilege $r_{\max} \in [s, s]$.

There is only one command, `unequal`, which can grant the right r_{notequal}, and by inspection, the privilege $r_{\text{notequal}} \in [s, s]$ may only be created if the privilege $r_{\text{greater}} \in [s, s]$ is present in the matrix. Therefore, in order to show the system safe for $r_{\text{notequal}} \in [s, s]$, it suffices to show the system safe for $r_{\text{greater}} \in [s, s]$.

There are two commands which grant the right r_{greater}: `addsubject` and `order`. We consider `addsubject` first. The command `addsubject` cannot grant $r_{\text{greater}} \in [s, s]$ other than to relate an existing subject to a new subject, and thus can only create a chain (or without $r_{\max}$, a tree) of subjects.

The command `order` can grant $r_{\text{greater}} \in [s, s]$ if and only if there exist principals s and s' such that $r_{\text{greater}} \in [s, s']$ and $r_{\text{greater}} \in [s', s]$. But the `order` command can only grant $r_{\text{greater}} \in [s', s]$ if there exists s'' such that $r_{\text{greater}} \in [s', s'']$ and $r_{\text{greater}} \in [s'', s]$. By induction, `order` can grant $r_{\text{greater}} \in [s, s]$ if and only if there exists a loop of subjects, each linked to the next by the right r_{greater}. However, `order` alone cannot create such a loop, it can only make it smaller.

Therefore the system is safe for $r_{\text{greater}} \in [s, s]$, and therefore the system is safe for $r_{\text{notequal}} \in [s, s]$, and consequently for $r_{\text{bang}} \in [s, s]$.

What we have demonstrated is a solution of the special safety problem for the protection system described in example 4.4. In this particular case, we may also show that the presence of an inequality is a stronger statement in the protection system than the absence of an equality. The presence of an inequality between two subjects is equivalent to the statement, "*This protection system is safe for the equality right between these two subjects*".

However, reducing a question to an instance of the safety problem does not necessarily help. The purpose of the protection system is not to decide the inequality of the subjects. The protection system is only a framework for proof in which the caller may prove the subjects to be unequal, as noted in section 2.3.1. In practice, the onus remains on the caller to establish the inequality of the two subjects involved by proving the system configuration unsafe for r_{notequal}. We must be able to find a sequence of commands which allows us (in this case) to explode the bomb, a stronger requirement than merely solving the safety problem to discover that such a sequence exists. Thankfully, in this case there is a simple and obvious algorithm for establishing an ordering relation in the matrix.

Example 4.6 (An Algorithm for the Inequality Map). Given two subjects s_0 and s_1, we assume that $s_0 \neq s_1$, and find a leaky computation which grants $r_{\text{notequal}} \in [s_0, s_1]$. Let $s_{\min}$ be the subject which holds the right $r_{\min}$ to itself: $r_{\min} \in [s_{\min}, s_{\min}]$. In other words, $s_{\min}$ is the first subject in the list of subjects each linked to the next by r_{next}: there is no s' such that $r_{\text{next}} \in [s', s]$. This subject is identifiable in linear time in $|S|$. Let $\text{next}(s)$ be the subject s' such that $r_{\text{next}} \in [s, s']$, in other words, s' is the subject after s in the list. This subject is identifiable at worst in linear time in $|S|$. The algorithm for computing the inequality map is then as described in algorithm 1.

Algorithm 1 Algorithm to perform leaky computation for $s' \in [r_{\text{notequal}}, s]$

```
Require: Two subjects, assuming $s_0 \neq s_1$. A least subject $s_{\min}$.
Ensure: $r_{\text{notequal}} \in [s_0, s_1]$
  $c \leftarrow s_{\min}$
  while $c \neq s_0$ and $c \neq s_1$ do
    $c \leftarrow \text{next}(c)$
  end while
  { Without loss of generality, we will assume that we found $s_0$. }
  repeat
    $c' \leftarrow \text{next}(c)$
    $\text{order}(s_0, c, c')$
  until $c' = s_1$
  $\text{unequal}(s_0, s_1)$
```

This algorithm executes in time polynomial in the index of s_1 in the linked list and proves the inequality of s_0 and s_1.

While completing the entire relation for an infinite number of subjects is not possible, for any two given subjects we can complete only a part of the relation allowing us to execute the **explode** command. We are not required to prove everything, only what we need to do the job.

4.8 Summary

We have reproduced the formalism of protection systems from [HRU76]. This model is sufficiently general to encapsulate most protection systems, but the original formalism is lacking any concept of responsibility or current principal.

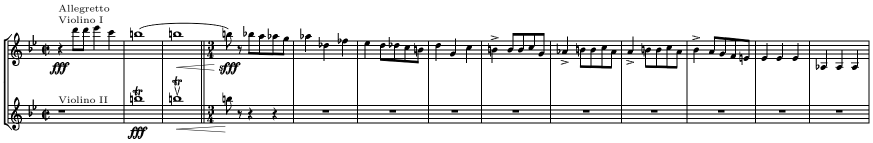
We have also learned that the general safety problem, to decide the safety of a given privilege in a given protection system, is undecidable. Thus it is undecidable in general whether a given protection system satisfies a policy, as discussed in section 3.

We have gained considerable insight into the formalism of protection systems and the solutions to specific cases of the safety problem by working through a specific example.

Material which follows from this section may be found in:

- Section 5: We extend our analytical abilities by learning to compute with protection systems. Section 5 is recommended to those willing to do battle with a little notation.
- Section 6: We study whether a restriction of the general model of protection systems might lead to a class of protection systems for which the safety problem is decidable.
- Section 7: Our development of practical protection systems combines the formalisms developed here with the business and policy requirements of sections 2 and 3. We aim to build useful protection systems which can be shown to satisfy policies.

Pay attention. Here comes the science bit . . .



5 Mathematical Properties of Protection Systems

There will arise many circumstances where we have to analyse a particularly nasty looking, or possibly just inconvenient protection system. It may be possible to analyse a simpler system which still contains the essence of the original system, and then somehow *transfer* our results to the more complex system. This is analogous to the way that problems are classified NP-Hard by reduction to a known NP-Hard problem. We therefore aim to identify relationships between protection systems such that computational properties are preserved by the relationship. We identify relationships of *simulation*, *equivalence* and *expressiveness*.

If one protection system can perform all the access control decisions of another, we say that the first system *simulates* the second. Simulation is an embedding of the simulated system into the simulating system, and thus preserves the existence of a leaky computation from the simulated protection system to the simulating protection system.

If two protection systems can each perform all the access control decisions possible within the other, then they are said to be *equivalent*. Equivalence is slightly stronger than a simple bisimulation; we require that the systems be truly isomorphic so that any reasoning about one system applies equally to the other. Thus any computation applying to one protection system will apply to any protection system to which it is equivalent.

We will produce a couple more useful mathematical tools. We can describe the *expressiveness* of a system in terms of the configurations reachable from a particular starting configuration. We also introduce mechanisms by which we may manipulate a protection system in ways external to the system without affecting the safety of the system. In this way, we may show that a system may sometimes be converted into a simpler or more powerful system while preserving the safety and expressiveness of the original system.

Notation.

Let Ψ be defined as in section 4.3.

Throughout, we will assume that $\Psi = (R, C)$, also $Q, Q' \in \mathrm{cf}(\Psi)$ and $c \in C$. Correspondingly, $\hat{\Psi} = (\hat{R}, \hat{C})$, $\hat{Q}, \hat{Q}' \in \mathrm{cf}(\hat{\Psi})$, $\hat{c} \in \hat{C}$. and so forth.

In general, we use decorations on maps as follows: $\phi : \mathrm{cf}(\Psi) \to \mathrm{cf}(\hat{\Psi})$, $\hat{\phi} : \mathrm{cf}(\hat{\Psi}) \to \mathrm{cf}(\Psi)$, and $\phi' : \mathrm{cf}(\hat{\Psi}) \to \mathrm{cf}(\bar{\Psi})$.

We will abuse the symbol $\vdash$ by writing $Q \vdash_{\Psi} Q'$ to mean that there is some command $c \in C$ such that $Q \vdash_c Q'$ as in section 4.7. Correspondingly, we write $\hat{Q} \vdash_{\hat{\Psi}} \hat{Q}'$, and so forth. The subscript serves to disambiguate the operands of the relation $\vdash$ when many protection systems are under discussion.

5.1 Simulation of Protection Systems

The mechanisms of a protection system permit and deny access to objects based on the instantaneous configuration of the system, rather than on the computation by which the configuration was reached. In order that a protection system $\hat{\Psi}$ may be said to simulate a system Ψ, it must be able to make the same access control decisions: there must be object protection code appropriately modified from that in example 4.2 on page 46 such that $\hat{\Psi}$ can make the same access decisions as Ψ. Therefore $\hat{\Psi}$ must have a set of configurations which contains, or at least contains an image of, the set of configurations of the simulated system. The definition of simulation is therefore centred around an injective map ϕ between configurations of the simulated system Ψ and configurations of the simulating system $\hat{\Psi}$.

Given such a map ϕ between configurations, we must also ensure that all the transformations of Ψ are all possible in $\hat{\Psi}$. A second map between command sets would immediately be over-restrictive; the mere existence of each and every transformation of Ψ in $\hat{\Psi}$ will suffice. Therefore, our definition of simulation is actually just an embedding of the relation $(\mathrm{cf}(\Psi), \vdash)$ into the relation $(\mathrm{cf}(\hat{\Psi}), \vdash)$.

Definition 5.1 (Simulation of Protection Systems). A protection system $\hat{\Psi}$ simulates a protection system Ψ if there exists a computable injective map $\phi : \mathrm{cf}(\Psi) \to \mathrm{cf}(\hat{\Psi})$ such that for any configuration $Q \in \mathrm{cf}(\Psi)$, if $Q \vdash Q'$ in Ψ then $\phi(Q) \vdash \phi(Q')$ in $\hat{\Psi}$. We then write $\hat{\Psi} \Rrightarrow \Psi$ via ϕ.

We may simply write $\hat{\Psi} \Rrightarrow \Psi$ if we do not need to make ϕ explicit. The reader may wish to think of $\hat{\Psi} \Rrightarrow \Psi$ as $\hat{\Psi} \supseteq \Psi$ since both the set of configurations of $\hat{\Psi}$ may be larger than that of Ψ and the set of commands $\hat{C}$ may be larger than those commands required to simulate commands in C from any configuration Q.

Simulation can be illustrated best by a commutative diagram; our diagram includes example command names c and $\hat{c}$.

$$\begin{tikzcd}
 & Q \arrow[r, "\phi"] \arrow[d, "c"] & \hat{Q} \arrow[d, "\hat{c}"] & \\
\Psi & & & \hat{\Psi} \\
 & Q' \arrow[r, "\phi"] & \hat{Q}' &
\end{tikzcd}$$

Given a configuration Q of a protection system Ψ, the map ϕ takes this configuration to a corresponding configuration $\hat{Q}$ of $\hat{\Psi}$. For any command c applied to Q, there is a command $\hat{c}$ in $\hat{\Psi}$ which makes the "*same*" transformation starting from $\hat{Q}$. Thus if we apply the command c to Q in Ψ, then map the new configuration into $\hat{\Psi}$, we get the same configuration as if we had mapped Q to $\hat{Q}$ via ϕ, then applied $\hat{c}$ in $\hat{\Psi}$ to reach $\hat{Q}'$.

We require the map ϕ to be injective.[18] Were it not so, then there would exist two configurations $Q, Q' \in \mathrm{cf}(\Psi)$ such that $\phi(Q) = \phi(Q')$. $\hat{\Psi}$ would not then be able to distinguish between the two configurations $\phi(Q)$ and $\phi(Q')$, and thus would not be able to make the same access decisions as Ψ, that is, simulate Ψ.

Theorem 5.1 (Properties of Simulation of Protection Systems). *Simulation is a reflexive and transitive relation.*

Proof.

Reflexivity: Let ϕ be the identity map, and let $\hat{c} = c$ always. Thus Ψ simulates itself.

Transitivity: Let $\bar{\Psi} \Rrightarrow \hat{\Psi}$ and $\hat{\Psi} \Rrightarrow \Psi$. We will show that $\bar{\Psi} \Rrightarrow \Psi$. The situation is illustrated thus:

$$\begin{tikzcd}
 & Q \arrow[r, "\phi"] \arrow[dd, "c"] & \hat{Q} \arrow[r, "\phi'"] \arrow[dd, "\hat{c}"] & \bar{Q} \arrow[dd, "\bar{c}"] & \\
\Psi & & & & \bar{\Psi} \\
 & Q' \arrow[r, "\phi"] & \hat{Q}' \arrow[r, "\phi'"] & \bar{Q}' &
\end{tikzcd}$$

The composition of two injective maps is injective. Therefore $\phi \circ \phi' : \mathrm{cf}(\Psi) \to \mathrm{cf}(\bar{\Psi})$ is an injection. Since $\hat{\Psi}$ simulates Ψ, for any $c \in C$ and $Q \in \mathrm{cf}(\Psi)$, there exists a $\hat{c} \in \hat{C}$ such that $\phi(Q) \vdash_{\hat{c}} \phi(Q')$. Given $\hat{Q} := \phi(Q)$, for any $\hat{c} \in \hat{C}$, and in particular the $\hat{c}$ defined above, there exists a $\bar{c} \in \bar{C}$ such that $\phi'(\hat{Q}) \vdash_{\bar{c}} \phi'(\hat{Q}')$. In other words, for any $Q \in \mathrm{cf}(\Psi)$, there exists a $\bar{c} \in \bar{C}$ such that $\phi'(\phi(Q)) \vdash_{\bar{c}} \phi'(\phi(Q'))$. Therefore, if $Q \vdash Q'$ then $\phi'(\phi(Q)) \vdash \phi'(\phi(Q'))$, and simulation is shown to be transitive.

□

Lemma 5.1 (Computability of Simulation). *Simulation is computable.*

Proof. By definition 5.1, ϕ is a computable map. Therefore, for any (Q, Q') such that $Q \vdash_c Q'$, we may compute $(\phi(Q), \phi(Q'))$. Since each $\hat{Q} \in \mathrm{cf}(\hat{\Psi})$ is finite, and the set $\hat{C}$ of commands is also finite, we may search $\hat{C}$ for a command $\hat{c}$ such that $\phi(Q) \vdash_{\hat{c}} \phi(Q')$.

□

It is assumed, if simulation is to be of any utility, that the map ϕ also preserves some useful semantics of the configurations. In particular, if Q_L is a configuration of Ψ such that a leak has occurred, then it is assumed that $\phi(Q_L)$ is a configuration of $\hat{\Psi}$ such that a corresponding leak has occurred in $\hat{\Psi}$. Given this semantic preservation, we may now translate instances of

[18] There are two critical differences between our definition of simulation and the definition of simulation from [Mil89]: firstly our map ϕ is injective, and secondly, $\hat{c}$ does not depend in any way upon c, where the definition of simulation in [Mil89] would require that $\hat{c} = c$. In our definition of simulation, $\hat{c}$ does not only depend on c but also on the configuration Q chosen. See example 5.2 for an equivalence where this is so.

the safety problem using ϕ: Let $\hat{\Psi} \Rrightarrow \Psi$ via ϕ. If (Q_0, Q_L) is an instance of the safety problem for Ψ where Q_0 is an initial state and Q_L a leak state then $(\phi(Q_0), \phi(Q_L))$ is a corresponding instance of the safety problem for $\hat{\Psi}$. By the definition of simulation, we may also translate leaky computations.

Theorem 5.2 (The Theorem of Simulation). *Let $\hat{\Psi} \Rrightarrow \Psi$. If there exists a leaky computation $Q_0 \vdash_{\Psi}^{*} Q_L$ in Ψ then there exists a leaky computation $\phi(Q_0) \vdash_{\hat{\Psi}}^{*} \phi(Q_L)$ in $\hat{\Psi}$.*

Proof. Let $Q_0 \vdash Q_1 \vdash \ldots \vdash Q_L$ be a leaky computation in Ψ. Then define $\hat{Q}_0 = \phi(Q_0)$, and so forth. By the assumption of the semantic properties of ϕ, $\phi(Q_L)$ is a leaky configuration of $\hat{\Psi}$. By the definition of simulation, each $\hat{Q}_i \vdash \hat{Q}_{i+1}$, thus we have a leaky computation in $\hat{\Psi}$: $\hat{Q}_0 \vdash \hat{Q}_1 \vdash \ldots \vdash \hat{Q}_L$.

□

While simulation preserves the property of safety in one direction, it does not preserve any properties regarding the decidability of the safety problem for the two systems: We have constructed simulation as a homomorphism of relations: an embedding of the image of the relation $\vdash$ in the new system; rather than as an isomorphism of the relation. In fact, the safety problem for the simulating system may have properties entirely unrelated to those of the simulated system. We will present a clarifying example.

Example 5.1. Let $\Psi = (R, C)$ where $C = \emptyset$. The safety problem for Ψ is trivial since Ψ has no commands. Either the configuration has a leak right, or it has not.

Let $\hat{\Psi} = (\hat{R}, \hat{C})$ where $\hat{R} = R$ and $\hat{C}$ is a set of commands such that $\hat{\Psi}$ emulates a Turing machine (as in the proof of theorem 4.1). Then $\hat{\Psi} \Rrightarrow \Psi$ via the identity map $\phi(Q) = Q$, but the safety problem for $\hat{\Psi}$ is undecidable, by theorem 4.1.

Let $\bar{\Psi} = (\bar{R}, \bar{C})$ where $\bar{R} = R$ and $\bar{C} = \hat{C} \cup \{\bar{c}\}$ where $\bar{c}$ enters a leak right from any configuration. Then $\bar{\Psi} \Rrightarrow \hat{\Psi}$ via the identity map $\phi'(Q) = Q$, but this time the safety problem for $\bar{\Psi}$ is trivial since it may leak from any configuration.

Thus simulation conveys no information about the relative decidability of the safety problems of the systems concerned.

5.2 Equivalence of Protection Systems

We would like to be able to divide protection systems into equivalence classes such that a result for the decidability of the safety problem for any one member of an equivalence class of protection systems would hold for all members of that class. We may then work with the simplest member of a class for any computations of decidability. For this purpose, we must define an equivalence relation on protection systems.

5.2.1 Introduction to Equivalence

If two protection systems are to be considered equivalent, then each must be able to make all the access control decisions possible within the other. The naïve formulation of equivalence would therefore be a bisimulation, but a true equivalence is actually slightly stronger than a bisimulation: it establishes an isomorphism between the two protection systems in such a way that reasoning about one system applies equally to the other. We can motivate this strengthening using the Schröder-Bernstein theorem.

Theorem 5.3 (Schröder-Bernstein Theorem ([Cam99])). *If there is an injective map from X to Y and an injective map from Y to X, then there is a bijective function from X to Y. In other words, if $|X| \leq |Y|$ and $|Y| \leq |X|$ then $|X| = |Y|$.*

The reader may refer to [Cam99] for two proofs of this theorem, one of which is reproduced in appendix A.2.

Theorem 5.4 (Bijection of Configurations). *Let $\hat{\Psi} \Rrightarrow \Psi$ via ϕ and $\Psi \Rrightarrow \hat{\Psi}$ via ϕ'. Then $|\operatorname{cf}(\Psi)| = |\operatorname{cf}(\hat{\Psi})|$.*

Proof. ϕ and ϕ' are injective maps. By the statement of the Schröder-Bernstein theorem there exists a bijection between $\operatorname{cf}(\Psi)$ and $\operatorname{cf}(\hat{\Psi})$, and thus the cardinality of $\operatorname{cf}(\Psi)$ is the same as the cardinality of $\operatorname{cf}(\hat{\Psi})$.

□

Since ϕ is expected to be a semantically preserving map, it is really very little strengthening of a simple bisimulation if we require that $\phi' = \phi^{-1}$.

Definition 5.2 (Equivalence of Protection Systems). Two protection systems Ψ and $\hat{\Psi}$ are equivalent if $\Psi \Rrightarrow \hat{\Psi}$ via ϕ and $\hat{\Psi} \Rrightarrow \Psi$ via ϕ^{-1} where ϕ is a bijection. We then write $\Psi \equiv \hat{\Psi}$ via ϕ.

In fact, given this definition of equivalence, the relation of *simulation* in definition 5.1 is a nonstrict partial order on all protection systems, with a minimal element being the protection system permitting nothing, and a maximal element being the protection system permitting everything.[19] Unlike simulation, in this case we do build a bijection between commandsets, making this more closely related to bisimulation from [Mil89]. We demonstrate this with a theorem: We may consider $(\operatorname{cf}(\Psi), \vdash_\Psi)$[20] and $(\operatorname{cf}(\hat{\Psi}), \vdash_{\hat{\Psi}})$ as relations, that is to say, directed graphs, and it is between these graphs that the isomorphism exists.

Theorem 5.5 (Isomorphism under Equivalence). *If $\Psi \equiv \hat{\Psi}$ then $Q \vdash_\Psi Q' \Leftrightarrow \phi(Q) \vdash_{\hat{\Psi}} \phi(Q')$.*

Proof. By definition 5.2, $\hat{\Psi} \Rrightarrow \Psi$ via ϕ. By definition 5.1, this means that $Q \vdash_\Psi Q' \Rightarrow \phi(Q) \vdash_{\hat{\Psi}} \phi(Q')$. Also, $\Psi \Rrightarrow \hat{\Psi}$ via ϕ^{-1}. Therefore, for any $\hat{Q}, \hat{Q}' \in \operatorname{cf}(\hat{\Psi})$, and specifically the $\phi(Q), \phi(Q')$ above, $\hat{Q} \vdash_{\hat{\Psi}} \hat{Q}' \Rightarrow \phi^{-1}(\hat{Q}) \vdash_\Psi \phi^{-1}(\hat{Q}')$.

□

[19] Unfortunately, we have Russell's paradox, a consequence of which is that the set of all sets cannot exist, and thus neither can the set of configurations of this maximal system.

[20] The subscript on $\vdash$ here serves to differentiate the relations in our notation.

5.2.2 Properties of Equivalence

Theorem 5.6 (Properties of Equivalence of Protection Systems). *The relation defined in definition 5.2 is an equivalence relation.*

Proof.

Reflexivity: Simulation is reflexive under the identity map. The identity map is a bijection. Equivalence is therefore reflexive.

Symmetry: Definition 5.2 is symmetric.

Transitivity: If $\Psi \equiv \hat{\Psi}$ via ϕ and $\hat{\Psi} \equiv \bar{\Psi}$ via ϕ' then $\Psi \Rrightarrow \bar{\Psi}$ via $\phi \circ \phi'$, which is a bijection. Also, $\bar{\Psi} \Rrightarrow \Psi$ via $\phi'^{-1} \circ \phi^{-1} = (\phi \circ \phi')^{-1}$. Therefore $\Psi \equiv \bar{\Psi}$.

□

Given a protection system and an initial configuration for that system, a corollary of the general safety problem (theorem 4.1) is that the set of reachable states of the system may be incomputable. However, we designed our definition of equivalence specifically to preserve the existence of computations over an equivalence, and thus to preserve the set of reachable configurations. We therefore have a corollary of the definition.

Corollary 5.1 (Isomorphism of Reachable Configurations). *If $\Psi \equiv \hat{\Psi}$ via ϕ, then the subgraph of $(\mathrm{cf}(\Psi), \vdash_\Psi)$ reachable from initial configuration Q is isomorphic to the subgraph of $(\mathrm{cf}(\hat{\Psi}), \vdash_{\hat{\Psi}})$ reachable from initial configuration $\phi(Q)$.*

Proof. The result is clear from the definition.

□

Theorem 5.7 (The First Theorem of Equivalence). *If $\Psi \equiv \hat{\Psi}$ via ϕ then the safety of $Q \in \mathrm{cf}(\Psi)$ is identical to the safety of $\phi(Q) \in \mathrm{cf}(\hat{\Psi})$.*

Proof. By theorem 5.2, if Ψ is leaky from Q, then $\hat{\Psi}$ is leaky from $\phi(Q)$. Thus, if $\hat{\Psi}$ is safe from $\phi(Q)$ then Ψ must be safe from Q. Correspondingly, if $\hat{\Psi}$ is leaky from $\phi(Q)$, then Ψ is leaky from Q. Thus, if Ψ is safe from Q then $\hat{\Psi}$ must be safe from $\phi(Q)$.

□

Note that in this proof we did not require that ϕ be a bijection.

Theorem 5.8 (The Second Theorem of Equivalence). *Let $\Psi \equiv \hat{\Psi}$ via ϕ. Then the safety problem for Ψ is decidable if and only if the safety problem for $\hat{\Psi}$ is decidable.*

Proof. By theorem 5.5, the graphs of the relations $(\mathrm{cf}(\Psi), \vdash_\Psi)$ and $(\mathrm{cf}(\hat{\Psi}), \vdash_{\hat{\Psi}})$ are isomorphic. By lemma 5.1, this isomorphism is computable, and thus any algorithm to manipulate $(\mathrm{cf}(\Psi), \vdash_\Psi)$ may equally manipulate $(\mathrm{cf}(\hat{\Psi}), \vdash_{\hat{\Psi}})$, and would achieve an equivalent result.

□

5.2.3 Examples of Equivalence

In our search for the simplest member of any equivalence class of protection systems, it may be to our advantage to construct a new member of the class. Equivalence provides us with a constructive means by which we may do so. We may define an appropriate bijection $\phi : \mathrm{cf}(\Psi) \rightarrow \mathrm{cf}(\hat{\Psi})$ and then by lemma 5.1, we may construct a second bijection $\nu : \vdash_\Psi \rightarrow \vdash_{\hat{\Psi}}$ (possibly from ϕ) to preserve the same semantics. In this example, we do indeed derive the transformation ν from the transformation ϕ, thus ensuring that definition 5.2 is satisfied. This is a useful technique which we will reuse in sections 6.1 and 6.4.

Example 5.2 (Equivalent Protection Systems). Let $\Psi = (R, C)$ with configurations (S, O, P). No command in C contains a *create subject*, *destroy subject*, *create object* or *destroy object* operation, thus S and O are some fixed constant sets and the matrix of Ψ is of a constant size. We will construct $\hat{\Psi} = (\hat{R}, \hat{C})$ with configurations $\hat{Q} = (\hat{S}, \hat{O}, \hat{P})$ such that $|\hat{S}| = 1$ and $|\hat{O}| = 1$. Let $|\hat{R}| = |S||O||R|$, specifically,

$$\hat{R} = \{\hat{r}_{s,o,r} \mid s \in S, o \in O, r \in R\}$$

$\hat{\Psi}$ will represent the presence or absence of each possible right in the fixed matrix of Ψ by the presence or absence of a Ψ-subject- and Ψ-object-specific right in the only cell in the matrix of $\hat{\Psi}$. For each triple $(s, o, r) \in S \times O \times R$ we designate a right $\hat{r}_{s,o,r}$, the presence (or absence) of which in the only cell of the matrix of $\hat{\Psi}$ will represent the presence (or absence) of the right $r \in [s, o]$ in the matrix of Ψ. Therefore, for any configuration Q of Ψ, there is a configuration $\hat{Q} = \phi(Q)$ of $\hat{\Psi}$ given by

$$\begin{aligned}
\hat{Q} &= (\hat{S}, \hat{O}, \hat{P}) \\
\hat{S} &= \{\hat{s}\} \\
\hat{O} &= \{\hat{o}\} \\
\hat{P} &= \{\hat{r}_{s,o,r} \mid r \in P[s, o]\}
\end{aligned}$$

Clearly, ϕ is a bijection.

For each command $c \in C$, we construct a set of commands $\hat{c}_{s,o,r}$ with each occurrence of $r \in [s, o]$ in any condition replaced by $\hat{r}_{s,o,r} \in [\hat{s}, \hat{o}]$. We also replace "**enter** r **into** $[s, o]$" with "**enter** $\hat{r}_{s,o,r}$ **into** $[\hat{s}, \hat{o}]$" and "**delete** r **from** $[s, o]$" with "**delete** $\hat{r}_{s,o,r}$ **from** $[\hat{s}, \hat{o}]$". For any configuration Q of Ψ and any command $c \in C$, there is now by construction a command $\hat{c} \in \hat{C}$ which has an equivalent effect on the matrix of $\hat{\Psi}$, and vice versa.

Clearly, $\Psi \equiv \hat{\Psi}$.

5.2.4 Restricted Equivalence

Simulation is too weak a relation to preserve any properties of the protection systems under consideration. But there are times when full equivalence is too strong. There is a middle ground, which we call '*restricted equivalence*' which preserves the computational niceties of full equivalence but without imposing a complete bijection between the protection systems under consideration.

Definition 5.3 (Restricted Equivalence). Two protection systems Ψ and $\hat{\Psi}$ are equivalent on a subset $A \subseteq \mathrm{cf}(\Psi)$ if there is a restriction $\phi|_A$ of ϕ such that

1. $\phi|_A$ is a bijection.
2. A is a closed subset of $\mathrm{cf}(\Psi)$ under the relation $\vdash$, that is, $Q \in A$ and $Q \vdash Q' \Rightarrow Q' \in A$.
3. $\phi|_A[A]$ is a closed subset of $\mathrm{cf}(\hat{\Psi})$ under the relation $\vdash$.
4. $\Psi \Rrightarrow \hat{\Psi}$ via $\phi|_A$ and $\hat{\Psi} \Rrightarrow \Psi$ via $\phi^{-1}|_{\phi[A]}$.

We then write $\Psi \equiv \Psi'$ on A via $\phi|_A$.

Restricted equivalence identifies a bijection between reachable sets of configurations, rather than between complete protection systems. Where previously, $\mathrm{cf}(\Psi)$ was closed under $\vdash$ by the definition of a $\mathrm{cf}(\Psi)$, we must now make this an element of the definition of restricted equivalence. However, the definition is otherwise almost identical to that of equivalence in definition 5.2, and this definition provides the same properties over the set A that the full definition of equivalence provides for $\mathrm{cf}(\Psi)$.

Having noted this, we may use restricted equivalence instead of equivalence in proofs, usually with $A = \mathrm{cf}(\Psi)$ but $\phi|_A[A] \subsetneq \mathrm{cf}(\hat{\Psi})$, In other words, ϕ is a bijection between $\mathrm{cf}(\Psi)$ and some closed, reachable subset of $\mathrm{cf}(\hat{\Psi})$. This saves us some considerable effort which would otherwise be required to limit $\mathrm{cf}(\hat{\Psi})$ in each case.

5.3 Expressiveness of Protection Systems

For many practical purposes, we may wish to ask of a protection system, "Can it do the job?" The definitions of simulation and equivalence in sections 5.1 and 5.2 show that we may specify that a protection system operates in a particular way. However, we may not be interested in the mechanisms of the protection system; only in which configurations may be reached by them. By analogy, it makes no difference whether we use Unix, Windows or VMS[21], as long as we can send SMTP mail.

The expressiveness of a protection system is a measure of the range of configurations which the protection system is capable of reaching, independently of the mechanism by which it reaches them.

[21] Only one, as long as the light bulb was made by DEC.

Definition 5.4 (Expressiveness of Protection Systems). A protection system $\hat{\Psi}$ is as expressive as a protection system Ψ if there exists an injective map[22] $\phi : \mathrm{cf}(\Psi) \to \mathrm{cf}(\hat{\Psi})$ such that for any configuration $Q \in \mathrm{cf}(\Psi)$, if $Q \vdash Q'$ in Ψ then $\phi(Q) \vdash^* \phi(Q')$ in $\hat{\Psi}$.[23]

A protection system $\hat{\Psi}$ simulates Ψ if there is a single command $\hat{c}$ in $\hat{C}$ such that $\phi(Q) \vdash_{\hat{c}} \phi(Q')$, in other words, $\hat{c}$ emulates c. If we allow a sequence of commands $\hat{c}_0, \hat{c}_1, \ldots$ to emulate each command c, we achieve a protection system which can reach a set of configurations which simulates that of Ψ, but by a nonequivalent mechanism. If every such sequence is of length 1, then $\hat{\Psi}$ simulates Ψ according to the stricter definition. Otherwise we say that $\hat{\Psi}$ is as expressive as Ψ because the set of reachable configurations of Ψ' *contains* an image of the set of reachable configurations of Ψ.

Frequently, we are not interested in the whole of $\mathrm{cf}(\Psi)$ since we only make access decisions based on some part of $\mathrm{cf}(\Psi)$. Many states in $\mathrm{cf}(\Psi)$ may be incidental to the policy, and serve only to allow the protection itself to operate. We will therefore give a definition of restricted expressiveness which limits the expression of the systems to some interesting or useful subset of $\mathrm{cf}(\Psi)$.

Definition 5.5 (Restricted Expressiveness of Protection Systems). A protection system $\hat{\Psi}$ is expressive as a protection system Ψ on a restricted set $A \subseteq \mathrm{cf}(\Psi)$ if there exists an injective map $\phi : \mathrm{cf}(\Psi) \to \mathrm{cf}(\hat{\Psi})$ such that for any configuration $Q \in A$, if $Q \vdash^* Q'$ in Ψ then $\phi(Q) \vdash^* \phi(Q')$ in $\hat{\Psi}$.

The whole idea of expressiveness is looser than that of simulation, and does not allow us to achieve the same computational results as either simulation or equivalence. However, it does allow us to make some powerful statements about the utility of particular protection systems, and is a more useful measure of the utility of a protection system when evaluated for practical purposes. The most useful protection system is likely to be one which allows every combination of rights permitted by policy to be entered into the matrix.

Restricted expressiveness will be used informally in section 7 to show that we lose no expressiveness when we derive a new system from an old system. But if we are to make transformations of systems and show that we lose no desired expressiveness, we must also show that we gain no undesired expressiveness; that is, we do not create any leaks in the system. For this, we need safety equivalence.

5.4 Safety Equivalence of Configurations

Within a particular protection system, there may be certain modifications we can make without affecting the safety or expressiveness of the system in any important way.[24] An example of such a modification might be the execution of any command by an untrusted principal; it produces a system under which the safety of every right with respect to that untrusted

[22] The assumption that ϕ preserves some semantics of $\mathrm{cf}(\Psi)$ is also made here.

[23] We might alternatively require $\phi(Q) \vdash^+ \phi(Q')$ but this does not seem necessary.

[24] That is, in any way that impinges upon policy.

principal is unchanged since that principal could have executed the command as a part of any leaky computation.

However, there may be other changes we may make to a configuration of a protection system without affecting the safety of any privileges. We might, for example, want to prove some form of equivalence of certain organisational structures by showing that each is as safe as the other. Thus we motivate the following definition.

Definition 5.6 (Safety Equivalence of Configurations). Let $Q, Q' \in \mathrm{cf}(\Psi)$.

A configuration $Q = (S, O, P)$ is safety equivalent to a configuration $Q' = (S', O', P')$ if the safety of $r \in [s, o]$ in Q is the same as the safety of $r \in [s, o]$ in Q' for all $s \in S \cap S', o \in O \cap O', r \in R$.

Notation. We use safety($r \in [s, o]$) to denote the safety of a privilege $r \in [s, o]$ in a system Ψ where Ψ is implicit. safety($. \in [., .]$) takes the values true and false where true > false.

Any extended study of a logic built around this definition would again be a considerable detour from the intended material of this thesis. However, these are interesting tools which we will put to some use in section 7, and we can outline some useful consequences of such a logic. For example, given the command

```
command upgrade($O$)
    let $s$ = current_principal
    if
        $r \in [s, O]$
    then
        enter $r'$ into $[s, O]$
end
```

then under all circumstances, the privilege $r' \in [s, o]$ is at most as safe as the privilege $r \in [s, o]$. We can express this as

$$\text{safety}(r' \in [s, o]) < \text{safety}(r \in [s, o])$$

since if the right r is unsafe, then the right r' is unsafe. This relation of safety is a property of the protection system Ψ, rather than any particular configuration of the system. The right r' might also be unsafe on its own, for example, if Ψ contains an unguarded command to grant r', in which case safety($r' \in [s, o]$) = false. Given a particular configuration of a system which is not safe for r', we might say that *in that configuration*, safety($r' \in [s, o]$) = false. This last is a property of a configuration of the system, over and above any properties of the system. The variables s and o might be bound in this last case, where previously they were always free.

This new logic of safety equivalence is conservative: It may pronounce a system unsafe when it is in fact safe. Consider, for example, the case of two mutually exclusive rights r and r' as manipulated by the following commands *bar* and *baz*. Given both rights, we may acquire the right r_{leak}, but since the rights are mutually exclusive in any configuration not initially containing both, no subject can never acquire the leak right to any object.

```
Command                                  Logical statement

command bar($S, O$)
    delete $r$ from $[S, O]$
                                         safety($r' \in [s, o]$) = false
    enter $r'$ into $[S, O]$
end
command baz($S, O$)
    delete $r'$ from $[S, O]$
                                         safety($r \in [s, o]$) = false
    enter $r$ into $[S, O]$
end
command qux($S, O$)
    if
        $r \in [S, O]$
                                         safety($r_{\text{leak}} \in [s, o]$) <
        $r' \in [S, O]$
                                               safety($r \in [s, o]$) $\vee$ safety($r' \in [s, o]$)
    then
        enter $r_{\text{leak}}$ into $[S, O]$
end
```

A resolution of these three statements gives safety($r_{\text{leak}} \in [s, o]$) = false as a property of the protection system and therefore for all possible configurations. This is untrue, since no principal can start from an empty configuration with no rights and gain both the rights r and r' to any object.

It is possible to define a conservative logic of safety for any protection system by translating the access matrix commands into relations as above. Such a logic will give false negative results for safety since the reduction to pure propositional logic throws away the modality information present in a computation. If this logic considers a privilege to be unsafe at any time, then it is considered always to be unsafe. However, in a monotonic system (as described in section 6.5) or a monooperational system (which may be made monotonic without increase of safety, as described in section 4.4), any leaked privilege stays leaked, and therefore this logic will give correct results.

In section 7, we manipulate monooperational systems by means external to those systems. We now have a tool which allows us to demonstrate that our manipulations have not affected the safety problem for the system using this logic.

5.5 Summary

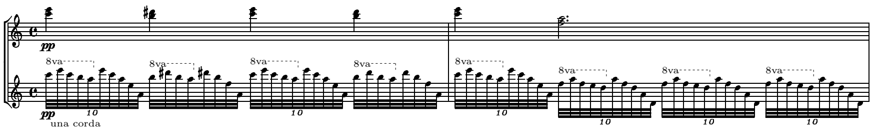
The main concepts we will carry forwards from this mathematical interlude are those of equivalence and expressiveness. We need equivalence for some of our decidability results, including those in sections 6.1 and 6.4. Expressiveness, on the other hand, is not used computationally. It first reappears as a requirement in section 7, whence it drives the choice of the protection system used in section 10.

Material which follows from this section may be found in:

- Section 6: We use our mathematical understanding to create more elegant proofs of the decidability or undecidability of various classes of protection system.

- Section 7: An understanding of the comparative behaviour of protection systems sometimes allows us to state in a practical scenario whether a system is better than another. An extension of this ability to compare systems allows us to develop new systems from old.

So how do we classify protection systems . . . ?



6 Classes of Protection System

The safety problem for a subclass of protection systems is a useful midway point between the general safety problem for all protection systems and the special safety problem for a particular implementation. It may useful, for example, to demonstrate a decidability result for all systems which only make one change to the matrix in each command. This is a more general classification of protection systems than that produced by the definition of equivalence in definition 5.2, since we only require certain particular properties of the systems in question rather than equivalence to a given protection system. This increased generality allows us to produce decidability results for considerably larger classes of protection system, and understand the impact of various properties which might hold of a protection system. We will demonstrate decidability results for the class safety problem for many of these classes; these results include existing proofs from [HRU76] and [HR78] in sections 6.3.2 and 6.5.2 and from [LB78] and [Bud83] in section 6.6.

Any polynomial time decidability result we discover has the potential to make available a wide range of protection systems for practical use and analysis. However, it is very hard to discover classes of protection system which are amenable to this form of analysis. There is little applicable rigorous method; the only known approach to identifying a class of protection systems with a decidable class safety problem is to hypothesise a likely looking class and attempt a proof. Furthermore, a positive decidability result does not imply the existence of an efficient algorithm, indeed we show that the general safety problem for some classes is decidable but complete on polynomial space.

We will call a protection system *decidable* if there exists a class containing that protection system with a decidable class safety problem. Unsurprisingly, even with the most severe restrictions of the general model suggested in this section, this approach fails to identify many rapidly decidable classes of protection system, and thus these results are for the most part of theoretical interest only. The ground-up approach of section 7 is of considerably more use for any practical purposes.

6.1 Bounded Systems

Many real protection systems may be shown equivalent to a some bounded protection system. Therefore, we commence our study with bounded systems. It is important to recognise the contribution of boundedness to the decidability of any safety problem. Intuitively, bounding the size of the configuration of a protection system reduces the problem space, and may allow us to use a search algorithm for the class safety problem. This is indeed so, and we have positive results for both subject- and object bounded systems.

6.1.1 Construction of Bounded Protection Systems

Definition 6.1 (Object Bounded Protection System). A B-object bounded protection system is a protection system in which the number of objects is bounded by some finite integer B.

$$\Psi \text{ is object bounded if } (S, O, P) \in \mathrm{cf}(\Psi) \Rightarrow |O| \leq B$$

Definition 6.2 (Subject Bounded Protection System). A B-subject bounded protection system is a protection system in which the number of subjects is bounded by some finite integer B.

$$\Psi \text{ is subject bounded if } (S, O, P) \in \mathrm{cf}(\Psi) \Rightarrow |S| \leq B$$

Since every subject is also an object, object boundedness is a strictly stronger constraint than subject boundedness. Any B-object bounded protection system is also a B-subject bounded protection system.

This definition of bounding is slightly artificial. We might naturally hypothesise a configuration for any system which breaks the bound. A more accurate definition would be, "*For any configurations Q and Q' such that Q is bounded by B and Q' is not, there is no command $c \in C$ such that $Q \vdash_c Q'$.*" In other words, while larger configurations exist, they are not reachable. However, our original definition better serves the intuition.

Any command in a bounded system which performs multiple operations must only create intermediate configurations which are themselves bounded. We may view this restriction of a protection system as an extra condition on the execution of a command: $\texttt{boundedp}(Q, c) \in \{\text{false}, \text{true}\}$, which is true if and only if the execution of c from Q will never cause the configuration of the protection system to exceed the implicit bound. We may execute the command c from Q only if $\texttt{boundedp}(Q, c) = \text{true}$. However, we may refactor this extra condition into our existing protection system for any finite bound by redesigning the commandset of our protection system such that it may never create a configuration which exceeds the bound B. This process is called '*bounding*'.

Definition 6.3 (Bounding of a Protection System). We may take a protection system and modify it to produce a system which is equivalent to the given system under all applications of commands which do not cause the configuration of the system to exceed the bound. This process of modification is called '*bounding*' and the new system is a '*bounding*' of the old system.

There are many reasons for a system to be bounded. In practice, every single protection system is limited by available memory, the size of the data type used to represent a subject, or some related issue, and this introduces an artificial bound on the number of subjects in the system. Other less artificial reasons for boundedness may include the nonexistence of any access matrix commands which create subjects or the choice of a set of access matrix commands which may only generate a bounded number of subjects from any given configuration.

Theorem 6.1 (Existence of Subject Bounded Protection Systems). *Any protection system Ψ may be modified into a B-subject bounded system $\hat{\Psi}$ where $\hat{\Psi}$ is a bounding of Ψ. The transformation uses a constant number of new rights and polynomial time and space.*

In brief, the proof proceeds as follows. All subjects are formed into a linked list with marked first and last elements, as in section 4.7.1. It is possible to write a guard clause which tests for the length of this list being equal to any constant. It is therefore possible to replace each command with a finite number of new commands, each of which will only execute when there are a given constant number of subjects existing. By replacing each command with such a set of new commands, we can control the maximum number of subjects in the matrix before the execution of every command and thus ensure that no command ever breaks the bound.

We must deal with the case of the configuration containing no subjects separately. We will represent such a configuration by a configuration containing exactly one subject with a discriminated right to itself for which we may test. We must then add to each set of new commands a command operating on a matrix with no subjects; this command must remove the special subject before creating any new subjects.

We now perform this proof formally.

Proof. Let $\Psi = (R, C)$. Let $\hat{\Psi} = (\hat{R}, \hat{C})$ where $\hat{R} = R \cup \{r_{\text{first}}, r_{\text{last}}, r_{\text{next}}, r_{\text{empty}}\}$. Let κ be an enumeration[25] of S. We define $\phi : \text{cf}(\Psi) \to \text{cf}(\hat{\Psi})$ by $\phi(Q) = \hat{Q} = (\hat{S}, \hat{O}, \hat{P})$ such that if $|S| = 0$ then $\hat{S} = \hat{O} = \{\hat{s}\}$ and $\hat{P}[\hat{s}, \hat{s}] = \{r_{\text{empty}}\}$, otherwise $\hat{S} = S$, $\hat{O} = O$, and

$$\hat{P}[s,o] = \begin{cases} P[s,o] \cup \{r_{\text{first}}\} & \text{if } o = s, \kappa(s) = 0, s = o \\ P[s,o] \cup \{r_{\text{next}}\} & \text{if } o \in S, \kappa(o) = \kappa(s) + 1 \\ P[s,o] \cup \{r_{\text{last}}\} & \text{if } o = s, \nexists s' : \kappa(s') = \kappa(s) + 1 \\ P[s,o] & \text{if none of the above hold} \end{cases}$$

Then ϕ is a bijection between configurations of Ψ where $|S| < B$ and a subset of the configurations of $\hat{\Psi}$. We must now construct a commandset for $\hat{\Psi}$ to satisfy definition 5.2. In

[25] See appendix A.4.

the course of its execution, any $c \in C$ will create and destroy subjects. We may compute over the execution of c

$$\delta_c = max_c(|\text{subjects created}| - |\text{subjects destroyed}|)$$

If c is executed from a matrix containing at most $B - \delta_c$ subjects, then no configuration following the execution of any operation of c will exceed the bound B on the number of subjects in the matrix. If $\delta_c > B$, then c can never be executed within the bounded system and we will ignore such c in the following construction. $\delta_c \geq 0$ since before any operations from c have been performed, $|\text{subjects created}| - |\text{subjects destroyed}| = 0$

For each $c \in C$, we may therefore create commands $\hat{c}_i \in \hat{C}$ as follows: If c is

```
command $\alpha(X_1, X_2, X_3, \ldots, X_k)$
    if
        $r_0 \in [X_{s_0}, X_{o_0}]$ and
        $r_1 \in [X_{s_1}, X_{o_1}]$ and
        ...
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        $\text{op}_1$
        $\text{op}_2$
        ...
        $\text{op}_n$
end
```

Then for each $i \in \{1, \ldots, B - \delta_c\}$ we create a command equivalent to α if and only if there are i subjects in the initial matrix.

```
command $\hat{\alpha}_{\mathrm{c}}^{(\mathrm{i})}(X_1, X_2, X_3, \ldots, X_k)$
    if
        $r_{\mathrm{first}} \in [S_0, S_0]$ and
        $r_{\mathrm{next}} \in [S_0, S_1]$ and
        $r_{\mathrm{next}} \in [S_1, S_2]$ and
        $\ldots$
        $r_{\mathrm{last}} \in [S_i, S_i]$ and
        $r_0 \in [X_{s_0}, X_{o_0}]$ and
        $r_1 \in [X_{s_1}, X_{o_1}]$ and
        $\ldots$
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        $\mathrm{op}_1$
        $\mathrm{op}_2$
        $\ldots$
        $\mathrm{op}_n$
end
```

We may also create a special case command which operates on a matrix containing no subjects.

```
command $\hat{\alpha}_{\mathrm{c}}^{(0)}(X_1, X_2, X_3, \ldots, X_k)$
    if
        $r_{\mathrm{empty}} \in [S, S]$ and
        $r_0 \in [X_{s_0}, X_{o_0}]$ and
        $r_1 \in [X_{s_1}, X_{o_1}]$ and
        $\ldots$
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        destroy subject $S$ if $\hat{\alpha}'$ creates a subject[26]
        $\mathrm{op}_1$
        $\mathrm{op}_2$
        $\ldots$
        $\mathrm{op}_n$
end
```

By construction, if $Q \vdash_c Q'$ then there is a command $\hat{c}$ such that $\phi(Q) \vdash_{\hat{c}} \phi(Q')$ as long as the bound B is not exceeded at any point in the execution of c. Also, for any command in $\hat{c}$ and a configuration $\hat{Q}$, there is a command $c \in C$ from which $\hat{c}$ was derived. c was constructed to be identical to $\hat{c}$ with the exception of checking the subject bound. Therefore, if $\hat{Q} \vdash_{\hat{c}} \hat{Q}'$, then $\phi^{-1}(\hat{Q}) \vdash_c \phi^{-1}(\hat{Q}')$. Therefore $\Psi \equiv \hat{\Psi}$ when $|S| < B$, as described in definition 5.2.

[26] This notation means that this primitive operation is included in the command if any subsequent operation is a **create subject** operation.

The number of new rights introduced is 4 so $|\hat{R}| = |R| + 4$. For each command, we construct at most $B+1$ new commands, so $|\hat{C}| \leq (B+1)|C|$. Therefore, the size of the new protection system and the time taken to construct it are given by

$$|\hat{\Psi}| = |\hat{R}| + |\hat{C}| \leq |R| + 4 + (B+1)|C|$$

□

It is not obvious that nontrivial object bounded protection systems exist within the formalism of section 4. The following theorem proves that these systems exist.

Theorem 6.2 (Existence of Object Bounded Protection Systems). *Every protection system Ψ may be modified into a B-object bounded system $\hat{\Psi}$ where $\hat{\Psi}$ is a bounding of Ψ. The transformation uses a constant number of new rights and polynomial time and space.*

This is another equivalence proof. First, we create a new protection system equivalent to the given protection system such that all objects are also subjects. We do this by tagging all subjects in the original system with a right to themselves saying that they are representing 'real' subjects from the original protection system. We then convert all remaining objects to subjects without this right. We modify all commands so that they create and destroy only subjects, and when using a subject as a subject, check that it is a "real" subject. Now that our protection system has no objects which are not also subjects, we can introduce a subject bound on this modified system as demonstrated in the proof of theorem 6.1. We will now perform this formally.

Proof. Let $\Psi = (R, C)$ be the given protection system. We will construct $\hat{\Psi} = (\hat{R}, \hat{C})$ equivalent to Ψ such that in any configuration $(\hat{S}, \hat{O}, \hat{P}) \in \mathrm{cf}(\hat{\Psi})$, $\hat{O} = \hat{S}$, in other words, there are no objects which are not also subjects. We may then continue with the transformation from the proof of theorem 6.1 to introduce a subject bound, which is then also an object bound on the original system.

Let $\hat{R} = R \cup \{r_{subject}\}$. We define a bijection ϕ as follows: $\phi((S, O, P)) = (O, O, \hat{P})$ where

$$\hat{P}[s,o] = \begin{cases} P[s,o] \cup \{r_{\text{subject}}\} & \text{if } s = o \\ P[s,o] & \text{if the above does not hold} \end{cases}$$

For each command in $c \in C$, we create a command $\hat{c} \in \hat{C}$ that verifies that each parameter from $\hat{O}$ used as a subject is a *real* subject of S. If c has the form

```
command $\alpha(X_1, X_2, X_3, \ldots, X_k)$
    if
        $r_0 \in [X_{s_0}, X_{o_0}]$ and
        $r_1 \in [X_{s_1}, X_{o_1}]$ and
        ...
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        $\mathrm{op}_1$
        $\mathrm{op}_2$
        ...
        $\mathrm{op}_n$
end
```

then $\hat{c}$ will be

```
command $\hat{\alpha}(X_1, X_2, X_3, \ldots, X_k)$
    if
        $r_{\text{subject}} \in [X_{s_0}, X_{s_0}]$
        $r_0 \in [X_{s_0}, X_{o_0}]$
        $r_{\text{subject}} \in [X_{s_1}, X_{s_1}]$
        $r_1 \in [X_{s_1}, X_{o_1}]$
        ...
        $r_{\text{subject}} \in [X_{s_m}, X_{s_m}]$
        $r_m \in [X_{s_m}, X_{o_m}]$
    then
        $\mathrm{op}_0$
        $\mathrm{op}_1$
        ...
        $\mathrm{op}_n$
end
```

Then we have a protection system $\hat{\Psi}$ equivalent to Ψ in which every configuration contains only subjects. We may now proceed to B-subject-bound this system as shown in the proof of theorem 6.1.

□

6.1.2 Decidability of the Class Safety Problem for Object Bounded Protection Systems

Theorem 6.3 (The Class Safety Problem for Object Bounded Systems). *The class safety problem is decidable for object bounded systems.*

The transformation defined by this proof is illustrated in figure 10.

	o_0	o_1	$\ldots$
s	r		
s'		r	

With multiple subjects

	o_0	o_1	$\ldots$
$\hat{s}$	r_s	$r_{s'}$	

With a single subject

Figure 10: The equivalence of B-subject and 1-subject systems

Proof. Let (R, C) be a B-object bounded protection system. Since $|S| \le |O| \le B$, there are at most

$$\sum_{i=0}^{B}\sum_{j=0}^{i} 2^{(ij|R|)} \le B^2 2^{B^2|R|}$$

configurations reachable by the system. These may be searched in polynomial space. ([Gur89]) □

This is only a slight generalisation of corollary 4.3 to the proof of the undecidability of the general safety problem in theorem 4.1, and by the construction in that proof, this problem may be shown to be complete on polynomial space.

6.1.3 Decidability of the Class Safety Problem for Subject Bounded Protection Systems

Theorem 6.4 (Equivalence of B-subject and 1-subject Systems). *There is a 1-subject bounded protection system equivalent to a given B-subject protection system.*

Proof. Let $\Psi = (R, C)$ be a B-subject bounded protection system with configurations $Q = (S, O, P)$. Without significant loss of generality, we will assume that $S \subseteq \{s_i \mid 0 \le i < B\}$. We will construct a new protection system $\hat{\Psi} = (\hat{R}, \hat{C})$ such that $\hat{\Psi} \equiv \Psi$ via ϕ (to be defined) and $\hat{\Psi}$ has configurations $\hat{Q} = (\hat{S}, \hat{O}, \hat{P})$ such that a configuration of $\hat{\Psi}$ contains only one subject: $\hat{S} = \{\hat{s}\}$.

For each right $r \in R$, $\hat{R}$ contains a set of rights r_i where the variable i discriminates the rights by the subject which held r in P. $\hat{R}$ also contains some new discriminated rights: ρ_{s_i} for identifying objects in $\hat{O}$ which represented subjects in S, $\rho_{\cancel{s}_i}$ for identifying subjects which may yet be created, and ρ_o for recording which possible subjects are not currently present in S:

$$\hat{R} = \{r_i \mid r \in R, 0 \le i < B\} \cup \{\rho_{s_i}, \rho_{\cancel{s}_i} \mid 0 \le i < B\} \cup \{\rho_o\}$$

Given $Q = (S, O, P) \in \mathrm{cf}(\Psi)$, we define the map $\phi : \mathrm{cf}(\Psi) \to \mathrm{cf}(\hat{\Psi})$ as follows: $\phi(Q) = (\hat{S}, \hat{O}, \hat{P})$ where

$$
\begin{aligned}
\hat{S} &= \{\hat{s}\} \\
\hat{O} &= O \cup \{\hat{s}\} \\
\hat{P}[\hat{s}, o] &= \begin{cases} \{\rho_{\not{s}_i} \mid s_i \notin S\} & \text{if } o = \hat{s} \\ \{r_i \mid r \in P[s_i, o]\} \cup \{\rho_{s_i}\} & \text{if } o = s_i \\ \{r_i \mid r \in P[s_i, o]\} & \text{if } o \in O \backslash S \end{cases}
\end{aligned}
$$

ϕ is clearly a bijection. We must now create a new set of commands for $\hat{C}$, an adaption of the commands $c \in C$ to the new right structure defined in $\hat{R}$, as we did in example 5.2. The subject variables of c are the variables X_i which are used as subjects, in other words, those which appear as the argument X in any of

$$
\begin{gathered}
r \in [X, o], \\
\textbf{enter } r \textbf{ into } [X, o] \\
\textbf{delete } r \textbf{ from } [X, o] \\
\textbf{create subject } X \\
\textbf{destroy subject } X
\end{gathered}
$$

Let χ_c be an enumeration of the distinct subject variables in c. For each command $c \in C$, for each possible choice of χ_c,[27] we create one new command in $\hat{C}$ by modifying c as follows: for each subject variable X_j in c, we must verify in $\hat{c}$ that the object represents a '*subject*' of $\hat{Q}$ by checking for $\rho_{s_{\chi_c(X_j)}} \in [\hat{s}, X_j]$. We must also adapt the existing checks for rights and operations as illustrated in the following table.

$r \in [X_s, X_o]$	becomes	$r_{\chi_c(X_s)} \in [\hat{s}, X_o]$
enter r **into** $[X_s, X_o]$	becomes	**enter** $r_{\chi_c(X_s)}$ **into** $[\hat{s}, X_o]$
delete r **from** $[X_s, X_o]$	becomes	**delete** $r_{\chi_c(X_s)}$ **from** $[\hat{s}, X_o]$

We illustrate with a comparison of the forms of commands in C and $\hat{C}$.

[27] A choice of χ_c is a composition of some fixed χ_c with a permutation.

The form of $c \in C$.

```
command $\alpha(X_1, X_2, X_3, \ldots, X_k)$
    if
        $\ldots$
        $r_i \in [X_{s_i}, X_{o_i}]$
        $\ldots$
    then
        $\ldots$
        $\mathrm{op}_i$
        $\ldots$
end
```

The form of $\hat{c} \in \hat{C}$.

```
command $\hat{\alpha}(X_1, X_2, X_3, \ldots, X_k)$
    if
        $\ldots$
        $\rho_{s_{\chi_c(X_{s_i})}} \in [\hat{s}, X_{s_i}]$
        $\ldots$
        $r_{\chi_c(X_{s_i})} \in [\hat{s}, X_{o_i}]$
        $\ldots$
    then
        $\ldots$
        $\mathrm{op}'_i$
        $\ldots$
end
```

We have shown how commands may be modified to support the modification of rights in the matrix. We may also support the creation and deletion of subjects by maintaining information about which '*subjects*' are represented by objects in $\hat{Q}$ by storing auxiliary information in $[\hat{s}, \hat{s}]$. Before creating a subject, we must check that there does not currently exist an object representing a given 'subject'. We do this by checking that

$$\rho_{\not{s}_i} \in [\hat{s}, \hat{s}]$$

is true before performing

```
create subject $X$
delete $\rho_{\not{s}_i}$ from $[\hat{s}, \hat{s}]$
enter $\rho_{s_i}$ into $[\hat{s}, X]$
```

After destroying the object representing a '*subject*' X, we must also

$$\textbf{enter } \rho_{\not{s}_{\chi_c(X)}} \textbf{ into } [\hat{s}, \hat{s}]$$

This use of extra rights for temporary storage is similar to the construction of theorem 6.1 on page 69.[28]

By construction, $\Psi \equiv \hat{\Psi}$ via ϕ.

□

It is important to realise that this transformation produces a new protection system $\hat{\Psi}$ the size of which is polynomial in B, not $|\Psi|$. Therefore, any decidability results relying upon this theorem as a part of the construction will not produce a result about time complexity in terms

[28] There is a very small caveat in that a command which repeatedly creates and destroys a subject will never exceed the bound B, but the choice of χ_c requires a different index for each created subject. This can be addressed in a similar manner to the proof of theorem 6.5. Alternatively, such temporary subjects may be elided from the command.

of $|\Psi|$.

Theorem 6.5 (The Class Safety Problem for Subject Bounded Systems). *The class safety problem is decidable for subject bounded systems.*

Proof. Let $\Psi = (R, C)$. By theorem 6.4, it suffices to consider a protection system with only one subject, therefore let $Q \in \mathrm{cf}(\Psi)$ where $Q = (S, O, P)$ and $S = \{\hat{s}\}$. Also by theorem 6.4, no operation in any command in C contains a *create subject* or *destroy subject* operation. We will temporarily ignore $P[\hat{s}, \hat{s}]$.

Objects and types:

Given any $o \in O$, the matrix P tells us only which rights are held to that object. There is no ordering in O or P, so P is just a list of subsets of R labelled by objects. Q therefore has the structure $P : O \to 2^R$.

The labels from O contain no relevant information, so we will discard them. We may now define an equivalence relation on O by

$$o \equiv o' \text{ if } P[\hat{s}, o] = P[\hat{s}, o']$$

P then divides O into $|2^R|$ equivalence classes. We will write $[A]$ for the equivalence class containing objects o such that $P(o) = A \subseteq R$ and call $[A]$ the '*type*' of o. The set of all types will be denoted $\mathbb{T}$, so $[A] \in \mathbb{T}$. Q therefore has the structure of a map $\xi : \mathbb{T} \to \mathbb{N}$.

The separated command set:

We now remove the requirement to consider aliasing of arguments in the invocation of a command. For any $c \in C$, let c have arity n_c.[29] When c is executed, the same argument might be passed for two or more of the n_c arguments of c. For each partition of the n_c arguments, we may identify and combine the instances of this argument within c, and thus generate a new '*separated command*' of c representing the calling of c possibly with the same values passed for more than one argument.

For any command c and any list $\bar{v}$ of n_c actual arguments for c, there is a separated command c_s of c and an ordered subset[30] $\bar{v}'$ of the arguments such that

$$Q \vdash_{c(\bar{v})} Q' \Longleftrightarrow Q \vdash_{c_s(\bar{v}')} Q'$$

In other words, for any list of arguments to c, there is a c_s which performs the same action as c given these arguments, but c_s takes only distinct arguments. We now define

$$C_s = \bigcup_{c \in C} \{c_s \mid c_s \text{ is a separated command of } c\}$$

The number of partitions of n_c elements is given by the Bell number for ${n_c}$[31] defined by the

[29] That is to say, c takes n_c arguments.

[30] By the nature of a set, the elements of the subset $\bar{v}'$ are distinct, where the elements of the list $\bar{v}$ were not necessarily distinct.

[31] See appendix A.3.

recurrence relation

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

The Bell numbers also clearly finite: there are a finite number of partitions of the n_c arguments. For each command $c \in C$, we have B_n commands in C_s. $|C|$ is finite, each n_c is finite, therefore $|C_s|$ is finite.

C_s is then called the '*separated command set*' of Ψ and may be used instead of the real command set, C. By construction, $\Psi = (R, C) \equiv (R, C_s)$.

The typed command set:

We will convert the separated command set C_s into a set of typed commands C_t such that the type of each argument to any $c_t \in C_t$ is fixed. For each $c_s \in C_s$, we may choose a list of possible types for the arguments of c_s. We may do this without fear of type collision, since commands in c_s are never called with aliased arguments. From the types of the arguments, we may immediately decide whether preconditions of the command c_s would be satisfied: $r \in [\hat{s}, o]$ if and only if o is of some type $[A]$ where $r \in A$. If the conditions would be satisfied, then we may generate a '*typed command*' c_t by removing the preconditions of the command and requiring instead that the arguments to c_t have the chosen types. We generate such a typed command c_t for each possible choice of types for the arguments of c_s.

For any separated command c_s and any ordered set $\bar{v}'$ of actual arguments for c_s such that the preconditions of c_s are satisfied, there exists a typed command c_t of c_s such that

$$Q \vdash_{c_s(\bar{v}')} Q' \Longleftrightarrow Q \vdash_{c_t(\bar{v}')} Q'$$

In other words, for any possible set of arguments which we could provide to c_s which would satisfy the preconditions of c_s, there is a typed command c_t which performs the same actions as c_s given any arguments of those types. We now define

$$C_t = \bigcup_{c_s \in C_s} \{c_t \mid c_t \text{ is a typed command of } c_s\}$$

Each c_s has n_{c_s} arguments. There are $|2^R|$ possible types, therefore we will generate at most $(n_{c_s})^{|2^R|}$ typed commands for c_s. This is clearly finite, and since C_s is finite, C_t is finite. C_t is then called the '*typed command set*' of Ψ and may be used instead of the separated command set C_s. Again, by construction, $\Psi = (R, C) \equiv (R, C_t)$.

The space of configurations:

Since objects are indistinguishable other than by their type, we may represent a configuration $Q \in \mathrm{cf}(\Psi)$ by a map $\xi : \mathbb{T} \to \mathbb{N}$, where $\xi([A])$ is the cardinality of the set of instances of the type $[A]$ in Q. Each ξ may be considered as a vector in the $2^{|R|}$-dimensional discrete space $\mathbb{N}^{2^{|R|}}$. We note that $\xi([A]) \geq 0$ for all $[A]$.

We define a several operations on the configuration vectors ξ elementwise:

$$\xi \leq \xi' \quad \text{if} \quad \xi([A]) \leq \xi'([A]) \quad \forall [A] \in \mathbb{T}$$

$$\xi = \xi' \quad \text{if} \quad \xi([A]) = \xi'([A]) \quad \forall [A] \in \mathbb{T}$$

$$(\xi + \xi')([A]) = \xi([A]) + \xi'([A])$$

$$(\xi - \xi')([A]) = \xi([A]) - \xi'([A])$$

$$|\xi| = \sum_{[A] \in \mathbb{T}} \xi([A])$$

Clearly, the operator $\leq$ is transitive, the operator $=$ is an equivalence relation and the operator $+$ is associative, each operator inheriting these properties from the classical operator used to define it. We will also define ξ_0 to be the element of the space such that $\xi_0([A]) = 0$ for any A. ξ_0 is then the minimal element of the space.

The vector command set:

We will make the semantics of each typed command precise in the space of configurations. We know precisely the types of all arguments to a typed command. We know that no argument is an alias for any other argument. We also know what is done to each of these arguments. We may therefore express every basic operation in a command in terms of the existence, creation and deletion of objects of known and fixed types.

Let $\gamma(c_t) : \mathbb{T} \rightarrow \mathbb{N}$ give the number of arguments of c_t with each type in $\mathbb{T}$. We note that $\gamma(c_t)([A]) \geq 0 \ \forall [A]$. Then for any configuration ξ and any typed command c_t, if $\gamma(c_t) \leq \xi$ then there exist distinct objects in the configuration ξ with suitable types to satisfy the preconditions of c_t.

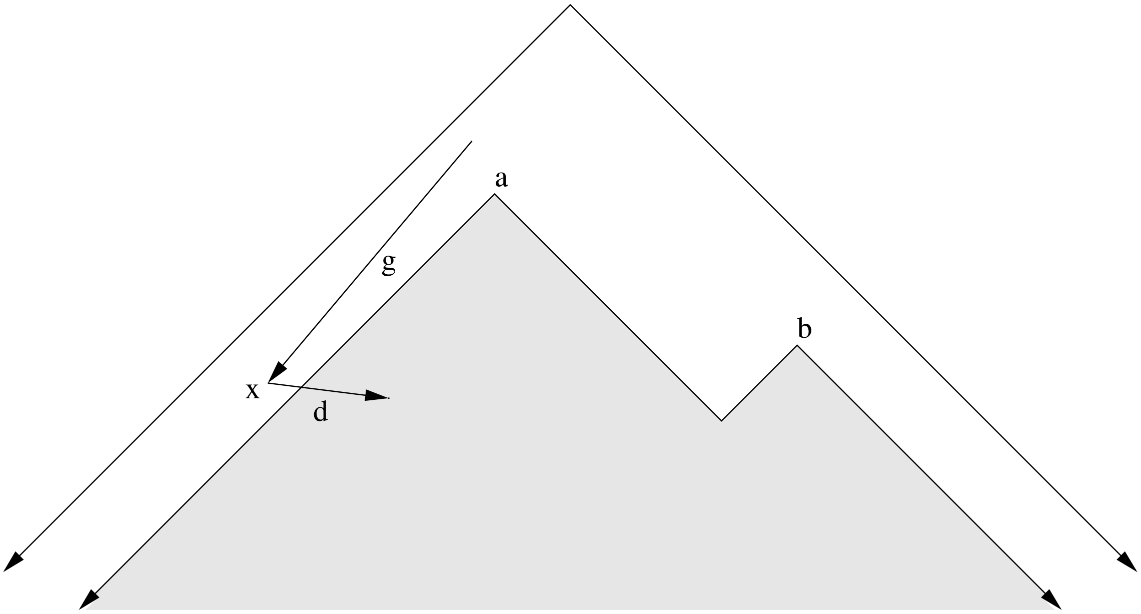
We may also compute the "*effect*" of c_t on the configuration ξ. Specifically, $\xi \vdash_{\text{op}} \xi'$ if either

1. op = create object $o : [\emptyset]$[32] , $\xi'([\emptyset]) = \xi([\emptyset]) + 1$; $\xi'([B]) = \xi([B])$ where $B \neq \emptyset$
2. op = destroy object $o : [A]$, $\xi'([A]) = \xi([A]) - 1$ if $\xi([A]) > 0$; $\xi'([B]) = \xi([B])$ where $B \neq A$
3. op = enter r into $[\hat{s}, o : [A]]$, $\xi'([A]) = \xi([A]) - 1$ if $\xi([A]) > 0$; $\xi'([B]) = \xi([B]) + 1$ where $B = A \cup \{r\}$; $\xi'([C]) = \xi([C])$ otherwise
4. op = delete r from $[\hat{s}, o : [A]]$, $\xi'([A]) = \xi([A]) - 1$ if $\xi([A]) > 0$; $\xi'([B]) = \xi([B]) + 1$ where $B = A \backslash \{r\}$; $\xi'([C]) = \xi([C])$ otherwise

An abstract interpretation of any command given the definitions above will compute an '*effect*' $\delta(c_t)$ of the command c_t such that if $\xi \vdash_{c_t} \xi'$, then $\delta(c_t) = \xi' - \xi$. The conditional execution of c_t may then be described thus:

$$\gamma(c_t) \leq \xi \Longleftrightarrow \xi \vdash_{c_t} (\xi + \delta(c_t))$$

[32] Any newly created object has no rights, and is thus of type $[\emptyset]$.



$$\Xi = \{a, b\} \qquad \xi = x$$
$$\gamma(c_t) = g \qquad \delta(c_t) = d$$

Figure 11: The space of configurations

The pair $(\gamma(c_t), \delta(c_t))$ is therefore a representation of the command c_t.

The leaky space:

We will address the class safety problem for some right r_{leak} by computing a space from which it is possible to reach a configuration under which $r_{\text{leak}} \in [\hat{s}, o]$ for any o using access matrix commands. We will call such configurations "*leaky configurations*" and the set of all such configurations the "*leaky space*". If our initial configuration ξ_i is within this leaky space, then it is unsafe for r_{leak}. If we can compute a maximal leaky space M^+,[33] and this space does not contain our ξ_i, then our initial configuration is safe for r_{leak}.

A command may test for the presence but not the absence of any number of instances of a type. Therefore, if ξ is a leaky configuration, then $\{\xi' \mid \xi \leq \xi'\}$ are all leaky configurations. Given some set Ξ of known leaky configurations, we may therefore describe the set of all known leaky configurations as

$$M(\Xi) = \bigcup_{\xi \in \Xi} \{\xi' \mid \xi \leq \xi'\}$$

We note that if $\xi, \xi' \in \Xi$ and $\xi \leq \xi'$ then $M(\Xi) = M(\Xi \setminus \{\xi'\})$, by the transitivity . of the $\leq$ operator on $\mathrm{cf}(\Psi)$. The set M is countable since it is isomorphic to a subset of $\mathbb{N}^{|2^R|}$. An argument similar to Cauchy's diagonal argument will allow us to count the elements of this space.[34]

[33] M is for 'mountain', as in figure 11.

[34] See theorem A.7 on page 219 where both arguments are presented.

We may identify an extension to a leaky space by identifying and executing a command c_t such that the following conditions hold.

1. $\xi \notin M(\Xi)$: ξ is not currently a known leaky configuration.
2. $\xi - \gamma(c_t) \geq \xi_0$: We may execute command c_t from ξ because the preconditions are satisfied.
3. $\xi + \delta(c_t) \in M(\Xi)$: The execution of the command c_t from ξ results in a configuration from which we may generate a leak.

We make the following natural definitions for the next part of the proof:

$$\begin{aligned} |\gamma(c_t)| &= \sum_{[A]\in\mathbb{T}} (\gamma(c_t))([A]) \\ |\delta(c_t)| &= \sum_{[A]\in\mathbb{T}} |(\delta(c_t))([A])| \end{aligned}$$

The finite search:

Given any c_t, we will show that the number of ξ that we must search in order to discover whether it is possible to satisfy conditions (1) - (3) above is countably finite. More specifically, we show that if

$$|\xi| > \max\left\{ \left(\max_{\xi' \in \Xi} \left\{ |\xi'| \right\} + |\delta(c_t)| \right), |\gamma(c_t)| \right\} \tag{1}$$

then there exists a $\hat{\xi}$ such that $|\hat{\xi}| < |\xi|$ and the conditions (1) - (3) are satisfied by $\hat{\xi}$. In other words, if ξ is a sufficiently large configuration, where '*sufficiently large*' is based on some '*size*' of the command c_t, then there is a smaller configuration $\hat{\xi}$ from which c_t also leaks. We define the shorthand L (for 'line') as in equation 1 above.

$$L = \max_{\xi' \in \Xi} \left\{ |\xi'| \right\} + |\delta(c_t)|$$

The set $\left\{ \xi \;\middle|\; |\xi| = L \right\}$ defines a cross section, a line across the space. The essence of the lemma is to state that if a command is executable from a configuration below this line, then it is executable on or above the line. Below the line, there are no interesting features of the subspace within reach of $\delta(c_t)$.

We included $\gamma(c_t)$ in equation (1) in case $|\gamma(c_t)| \geq L$. By definition, any ξ from which we may execute c_t must satisfy $|\xi| \geq |\gamma(c_t)|$ thus we must allow at least that $|\xi| = |\gamma(c_t)|$. We may assume that $|\gamma(c_t)| < L$ since by the transitivity of $\leq$, we may add a new $\bar{\xi}$ to Ξ such that $|\bar{\xi}| > |\gamma(c_t)|$.[35] Therefore, without loss of generality, we may ignore $|\gamma(c_t)|$ as a part of the condition on ξ in the rest of this proof.

[35] See the note about Ξ on page 80.

It remains to show that

$$\left.\begin{array}{c} \xi \notin M(\Xi) \\ \xi + \delta(c_t) \in M(\Xi) \\ |\xi| > L \end{array}\right\} \Longrightarrow \exists\hat{\xi} : \left\{\begin{array}{c} \hat{\xi} \notin M(\Xi) \\ \hat{\xi} + \delta(c_t) \in M(\Xi) \\ |\hat{\xi}| < |\xi| \end{array}\right.$$

then we may construct a $\hat{\xi}$ such that $|\hat{\xi}| \leq L$ by induction. We will in fact construct a $\hat{\xi} < \xi$.[36]

Let such a ξ be given. Since $\xi + \delta(c_t) \in M(\Xi)$, we may choose a representative $\xi_\Xi \in \Xi$ such that $(\xi + \delta(c_t)) > \xi_\Xi$. Thus $(\xi + \delta(c_t)) \in M(\{\xi_\Xi\}) \subseteq M(\Xi)$ but $\xi \notin M(\Xi) \supseteq M(\{\xi\})$. Let:

$$\begin{array}{rcl} \mathbb{A} & = & \{[A] | \xi([A]) = 0\} \\ \mathbb{B} & = & \{[A] | \xi([A]) > 0 \text{ and } (\xi + \delta(c_t))([A]) = \xi_\Xi([A])\} \\ \mathbb{C} & = & \{[A] | \xi([A]) > 0 \text{ and } (\xi + \delta(c_t))([A]) > \xi_\Xi([A])\} \\ \mathbb{T} & = & \mathbb{A} \oplus \mathbb{B} \oplus \mathbb{C} \end{array}$$

$\mathbb{A}$ is a strict subset of $\mathbb{T}$, otherwise $|\xi| = 0 < L$, which is a contradiction. Therefore $\mathbb{B} \oplus \mathbb{C}$ is not empty. Also, $\mathbb{C}$ is not empty. This is harder to show. For all $[A] \in \mathbb{B}$:

$$\begin{array}{rcl} (\xi + \delta(c_t))([A]) & = & \xi_\Xi([A]) \\ \xi([A]) + \delta(c_t)([A]) & = & \xi_\Xi([A]) \\ \xi([A]) & = & \xi_\Xi([A]) - \delta(c_t)([A]) \end{array}$$

Therefore

$$\sum_{[A]\in\mathbb{A}} \xi([A]) = 0 \tag{2}$$

$$\sum_{[A]\in\mathbb{B}} \xi([A]) = \left(\sum_{[A]\in\mathbb{B}} \xi_\Xi([A]) - \sum_{[A]\in\mathbb{B}} \delta(c_t)([A])\right) \tag{3}$$

Now, by equation 1:

$$|\xi| > |\xi_\Xi| + |\delta(c_t)| \tag{4}$$

[36] $\hat{\xi} < \xi$ means $\hat{\xi} \leq \xi$ and $\exists [A]$ such that $\hat{\xi}([A]) < \xi([A])$, equivalently, $\hat{\xi} \leq \xi$ and $\hat{\xi} \neq \xi$.

And when we subtract equations 3 and 2 from inequality 4, we get:

$$\begin{aligned}
|\xi| - \left(\sum_{[A]\in\mathbb{B}} \xi([A])\right) - \left(\sum_{[A]\in\mathbb{A}} \xi([A])\right) &> (|\xi_\Xi| + |\delta(c_t)|) - \left(\sum_{[A]\in\mathbb{B}} \xi_\Xi([A]) + \sum_{[A]\in\mathbb{B}} \delta(c_t)([A])\right)\\
|\xi| - \sum_{[A]\in\mathbb{A}} \xi([A]) - \sum_{[A]\in\mathbb{B}} \xi([A]) &> \left(|\xi_\Xi| - \sum_{[A]\in\mathbb{B}} \xi_\Xi([A])\right) + \left(|\delta(c_t)| - \sum_{[A]\in\mathbb{B}} \delta(c_t)([A])\right)\\
\sum_{[A]\in\mathbb{C}} \xi([A]) &> \sum_{[A]\in\mathbb{A}\oplus\mathbb{C}} \xi_\Xi([A]) + \sum_{[A]\in\mathbb{A}\oplus\mathbb{C}} \delta(c_t)([A])
\end{aligned}$$

Now if $\mathbb{C}$ is empty, then the left hand side is zero, while the right hand side is clearly not zero. Therefore $\mathbb{C}$ is not empty, and thus there exists $[A] \in \mathbb{C}$ such that $\xi([A]) > 0$ and $(\xi+\delta(c_t)) > \xi_\Xi([A])$. Now we may construct $\hat{\xi}$ such that $\hat{\xi}([A]) = \xi([A])-1$ and $\hat{\xi}([A']) = \xi([A'])$ otherwise. By induction we may now generate a $\hat{\hat{\xi}}$ such that $|\hat{\hat{\xi}}| \le L$.

Reintroducing $P[\hat{s}, \hat{s}]$:

There are two ways in which $P[\hat{s}, \hat{s}]$ may be reintroduced. First, we might instead solve the safety problem for the equivalent system which uses a discriminated object to represent $\hat{s}$, using one new subject to which no rights may ever be granted. Otherwise, we might compute a maximal leaky space for each possible $P[\hat{s}, \hat{s}] \in 2^R$ iteratively. We feel that both of these techniques have been covered earlier, and that this proof is already more than long enough.

The algorithm:

We will exhibit an algorithm to compute the maximal leaky space of a given protection system.

1. Input $\Psi = (R, C)$, a 1-subject bounded protection system.
2. Input $Q = (\{\hat{s}\}, O, P)$, a configuration of Ψ.
3. Compute C_s, the separated command set of Ψ.
4. Compute C_t, the typed command set of Ψ.
5. Compute ξ_i, the vector representation of Q.
6. Let $\mathbb{T}_{\text{leak}} = \{[A] \mid r_{\text{leak}} \in A\}$, $\mathbb{T}_{\text{leak}}$ is the set of types of objects to which the right r_{leak} has been leaked.
7. Compute $\Xi = \bigcup_{[A]\in\mathbb{T}_{\text{leak}}} \{\xi \mid \xi([A]) = 1, \xi([B]) = 0 \text{ where } B \neq A\}$, in other words, Ξ is the set of configurations with exactly one object o where o has a type $[A]$ such that $r_{\text{leak}} \in A$: in classical language, $r_{\text{leak}} \in [\hat{s}, o]$. A leak has already occurred.
8. For each c_t, attempt to extend the leaky space using c_t from all configurations ξ such that $|\xi| < L_\Xi$.
9. When the leaky space may not be extended by any c_t, the maximal leaky space has been computed.

□

Thus the class safety problem for subject bounded protection systems is decidable.

6.2 Monoconditional Protection Systems

A monoconditional system is a system within which each access matrix command has only one test in its condition. We suggested in section 4.5.4 that object protection code may test for only a single right without loss of generality, and with the requirement in section 2.3.3 that the protection system must protect itself, it is only fair to guess that it might do so with a single right.

It is therefore perhaps a surprise that the class of monoconditional protection systems has not been studied before, given that it is one of very few classes of protection system with a simple, low order polynomial decidability result. However, in practice, truly monoconditional systems are little used due to their lack of expressiveness.[37] Most of the practical protection systems in section 7 are biconditional at most, but this biconditionality gives them the power to express arbitrarily complex systems. The following results are therefore of mainly theoretical interest.

6.2.1 Construction of the Monoconditional Protection System

Definition 6.4 (Monoconditional Access Matrix Command). An access matrix command is monoconditional if the guard clause contains at most one test for a right.

Definition 6.5 (Monoconditional Protection System). A protection system is monoconditional if all commands within it are monoconditional.

6.2.2 Decidability of the Class Safety Problem for Monoconditional Protection Systems

Theorem 6.6 (Decidability of the Class Safety Problem for Monoconditional Protection Systems). *There is an algorithm which decides whether a given monoconditional protection system is unsafe for a given generic right r.*

Proof. A monoconditional access matrix command can only test either of $r \in [s, s]$ or $r \in [s, o]$ (where $s \neq o$). It may perform no further tests on either s or o. In the first case, no restrictions may be made on s other than that it hold r to itself, and therefore $r \in [s, s]$ is equivalent to "*Does the right r occur anywhere on the long diagonal of the matrix?*". In the second case, s and o are independent and may still be equal, and therefore $r \in [s, o]$ is equivalent to, "*Does the right r occur anywhere in the matrix?*"

This simplicity allows the length of the shortest leaky computation to be bounded. Suppose

$$Q_0 \vdash_{c_1} Q_1 \vdash_{c_2} \ldots \vdash_{c_m} Q_m$$

[37] In the non-formal sense, that is, not as defined in section 5.3.

is a minimal leaky computation.

First, we claim that c_m tests for a right entered into the matrix by c_{m-1} (and not subsequently removed by the same command). If not, then we could form a shorter computation

$$Q_0 \vdash_{c_1} Q_1 \vdash_{c_2} \ldots \vdash_{c_{m-1}} Q_{m-2} \vdash_{c_m} Q'_m$$

Since the command c_m can rely only on the existence, rather than the nonexistence, of the right in the matrix, and that right was not entered by c_{m-1}, the right must have been present in configuration Q_{m-1} and this we may still execute c_m. Since c_m created the leak, the computation is still leaky. By reapplying this logic to c_{m-1}, c_{m-2}, and so on, we discover that each command must rely only on a right entered into the matrix by the immediately previous command.

Second, we claim that each right r is required to be present in the matrix at most twice. Suppose not, and let

$$Q_0 \vdash_{c_1} \ldots \vdash Q_{i-1} \vdash_{c_i} Q_i \vdash \ldots \vdash Q_{j-1} \vdash_{c_j} Q_j \vdash \ldots \vdash Q_{k-1} \vdash_{c_k} Q_k \vdash \ldots \vdash_{c_m} Q_m$$

be a computation where c_i, c_j and c_k all test for the presence of some right r in the matrix. Recall that the only possible forms for a guard clause in a monoconditional access matrix command are $r \in [s, o]$, that a right is required to be present in the matrix, and $r \in [s, s]$, that a right is required to be present on the long diagonal of the matrix. We note that if a configuration satisfies $r \in [s, s]$ then it satisfies $r \in [s, o]$, that is, if a right is present on the long diagonal, then it allows the execution of any command requiring that right.

- If c_i requires the right r to be present on the long diagonal, then the condition of c_k is satisfied by Q_{i-1} and we may form a shorter computation

  $$Q_0 \vdash_{c_1} \ldots Q_{i-1} \vdash_{c_k} Q'_k \vdash \ldots \vdash_{c_m} Q'_m$$

  Since c_m creates a leak, the computation is still leaky.

- If c_i requires the right r to be present but not necessarily on the long diagonal and c_j requires the right to be present on the long diagonal, then similarly the condition of c_k is satisfied by Q_{j-1} and we may form a shorter leaky computation

  $$Q_0 \vdash_{c_1} \ldots Q_{j-1} \vdash_{c_k} Q'_k \vdash \ldots \vdash_{c_m} Q'_m$$

- If c_i requires the right r to be present but not necessarily on the long diagonal and c_j requires the right to be present but also not necessarily on the long diagonal, then the condition of c_j is satisfied by Q_{i-1} and we may form a shorter leaky computation

  $$Q_0 \vdash_{c_1} \ldots Q_{i-1} \vdash_{c_j} Q'_j \vdash \ldots \vdash_{c_m} Q'_m$$

Therefore, the shortest leaky computation tests for each right at most twice, and the length of this shortest leaky computation is bounded by $2|R|$. A procedure trying all sequences of commands of maximum length $2|R|$ will identify any possible leaky computation.

□

Actually, we can do a little more than just bound the length of the maximal leaky computation. We have sufficient insight into the nature of such a computation for us to show the search process to be equivalent to a search for a path in a graph of $2|R|$ nodes. Each node represents the existence of a particular right either on or off the long diagonal of the matrix. Edges in the graph are given by the existence of a command c which permits a transformation between the states represented by the nodes. The starting node is the node representing a leak, and the target node is any node satisfied by the initial configuration. Therefore, a relatively fast algorithm is possible for deciding safety.

6.3 Monooperational Protection Systems

The proof of the decidability of the class safety problem for monooperational protection systems was introduced together with the definitions in section 4 by [HRU76]. The authors of that paper did not have much of our framework available to them, and thus the argument is constructed very elegantly from the bare bones of the definition. It is reproduced in this section.

The result shows that the safety problem for monooperational protection systems is decidable, but is complete in NP time. While this is not directly of practical use, it is interesting to note that all the practical protection systems in section 7 are monooperational, and decide safety using a search algorithm similar to that proposed here.

6.3.1 Construction of the Monooperational Protection System

Definition 6.6 (Monooperational Access Matrix Command). An access matrix command is monooperational if it contains a single primitive operation.

Definition 6.7 (Monooperational Protection System). A protection system is monooperational if all commands within it are monooperational.

6.3.2 Decidability of the Class Safety Problem for Monooperational Protection Systems

The following theorem and proof are reproduced from [HRU76] with some minor updates to the notation for our framework.

Theorem 6.7 (Decidability of the Class Safety Problem for Monooperational Protection Systems ([HRU76])). *There is an algorithm which decides whether a given monooperational protection system is unsafe for a given generic right r.*

Proof. The proof hinges on two simple observations. First, commands can test for the presence of rights, but not for the absence of rights or objects. This allows delete and destroy commands[38] to be removed from computations leading to a leak. Second, a command can only identify objects by the rights in their row and column of the access matrix. No monooperational command can both create an object and enter rights, so multiple creates can be removed from computations, leaving the creation of only one subject. This allows the length of the shortest '*leaky*' computation to be bounded.

Suppose

$$Q_0 \vdash_{c_1} Q_1 \vdash_{c_2} \ldots \vdash_{c_m} Q_m \tag{5}$$

is a minimal length computation reaching some configuration Q_m for which there is a command α leaking r. Let $Q_i = (S_i, O_i, P_i)$. Now we claim that c_i, $2 \leq i \leq m$ is an enter command, and c_1 is either an enter or create subject command. Suppose not, and let c_n be the last non-enter command in the sequence (5). Then we could form a shorter computation

$$Q_0 \vdash_{c_1} Q_1 \vdash \ldots Q_{n-1} \vdash_{c'_{n+1}} Q'_{n+1} \vdash \ldots \vdash_{c'_m} Q'_m$$

as follows.

1. If c_n is a delete or destroy command, let $c'_i = c_i$ and $Q'_i = Q_i$ plus the right, subject or object which would have been deleted or destroyed by c_n. By the first observation above, c_i cannot distinguish Q_{i-1} from Q'_{i-1}, so $Q'i - 1 \vdash_{c'_i} Q'_i$ holds. Likewise, α leaks r from Q'_m since it did from Q_m.

2. Suppose c_n is a create subject command and $|S_{n-1}| \geq 1$ or c_n is a create object command. Note that α leaks r from Q_m by assumption, so α is an enter command. Further, we must have $|S_m| \geq 1$ and
$$|S_m| = |S_{m-1}| = \ldots = |S_n| \geq 1$$
($c_m, \ldots, c_{n+1}$ are enter commands by assumption). Thus $|S_{n-1}| \geq 1$ even if c_n is a create object command. Let $s \in S_{n-1}$. Let o be the name of the object created by c_n. Now we can let $c'_i = c_i$ with s replacing all occurrences of o, and $Q'_i = Q_i$ with s and o merged. For example, if $o \in O_n - S_n$ we would have
$$S'_i = S_i$$
$$O'_i = O_i - \{o\}$$
$$P'_i[x,y] = \begin{cases} P_i[x,y] & \text{if} \quad y \neq s \\ P_i[x,s] \cup P_i[s,o] & \text{if} \quad y = s \end{cases}$$

[38] Since the system is monooperational, we can identify the command by the type of the primitive operation contained within it.

Clearly,

$$P_i[x, o] \subseteq P'_i[x, s]$$

so for any condition in c_i satisfied by o, the corresponding condition in c'_i is satisfied by s. Likewise for the conditions of α.

3. Otherwise, we have $|S_{n-1}| = 0$, c_n is a create subject command, and $n \geq 2$. The construction in this case is slightly different — the create subject command cannot be deleted (subsequent "enters" would have no place to enter into). However, the commands preceding c_n can be skipped (provided that the names of objects created by them are replaced), giving
$$Q_0 \vdash_{c_n} Q'_n \vdash_{c'_{n+1}} Q'_{n+1} \vdash \ldots \vdash Q'_m$$
where, if $S_n = s$, we have c'_i is c_i with s replacing the names of all objects in O_{n-1}, and Q'_i is Q_i with s merged with all $o \in O_{n-1}$.

In each of these cases, we have created a shorter "leaky" computation, contradicting the supposed minimality of (5). Now we note that no c_i enters a right r into a cell of the access matrix already containing r, else we could get a shorter sequence by deleting c_i. Thus we have an upper bound on m:

$$m \leq g(|S_0| + 1)(|O_0| + 1) + 1$$

where g is the number of generic rights.

An obvious decision procedure now presents itself — try all sequences of enter commands, optionally starting with a create subject command, of length up to the bound given above. This algorithm has time complexity exponential in the size of the matrix.

□

Corollary 6.1 (NP-Hard Status of the Decision Algorithm ([HRU76])). *We can give a short proof that a given configuration of a given monooperational protection system is* not *safe by giving a leaky computation (as in equation 5 on page 87), although this computation may be difficult to find.*

However, it is likely that there is no proof system in which we can guarantee existence of, let alone find, short proofs that a given configuration of a given monooperational protection system is *safe.*[39]

6.4 Boolean Protection Systems

We have made restrictions of the full protection system formalism such that commands may contain only one test for a right or only one operation. The boolean protection system is another computationally simpler version of the access matrix system in which there is only one generic right.

[39] Under the assumption that $P \neq NP$. This is likely.

Our starting point for the construction of a practical protection system in section 7 is a boolean, monooperational system, which we show has a safety problem decidable in linear time. However, this positive result does not hold for the general case of a boolean protection system. We will show that this simplified system is as powerful as the full formalism and thus that the class safety problem for boolean protection systems is undecidable.

6.4.1 Construction of the Boolean Protection System

Definition 6.8 (n-right Protection System). An n-right protection system is a system $\Psi = (R, C)$ such that $|R| = n$.

Definition 6.9 (Boolean Protection System). A boolean protection system is a 1-right protection system.

The boolean protection system is constructed from the original formalism for protection systems in section 4.3, but with only one right. $|R| = 1$. We will call this right r. The cell in the matrix denoted $[s, o]$ either contains the only right r or does not contain r. Equivalently, we might say that $[s, o]$ contains a boolean value which is *true* if s has the right to access o, and *false* otherwise, hence the term "*boolean protection system*".

The right r expresses a single relation between subjects and objects: $r \in [s, o]$ may be considered as the binary relation $s \succ o$. As mathematicians, we have considerable experience with such relations. A few examples of commands are described below; others are of course possible.

(1): Transitive relation

```
command transitivity($O, P$)
    let $s$ = current_principal
    if
        $r \in [s, O]$ and
        $r \in [O, P]$
    then
        enter $r$ into $[s, P]$
end
```

(3): Identify subjects

```
command identify_subjects($T, O$)
    let $s$ = current_principal
    if
        $r \in [T, O]$
    then
        enter $r$ into $[s, O]$
end
```

(2): Symmetry in o and s

```
command symmetry($O$)
    let $s$ = current_principal
    if
        $r \in [O, s]$
    then
        enter $r$ into $[s, O]$
end
```

(4): Identify objects

```
command identify_objects($O, T$)
    let $s$ = current_principal
    if
        $r \in [s, O]$
    then
        enter $r$ into $[s, T]$
end
```

Any choice of such boolean access matrix commands is equivalent to a choice of axioms for the relation on S. We may recognise from set theory the structure that would be produced in eventuality by repeated applications of these rules. By selecting some subset of these simple commands, we might establish a useful protection system which we may then study.[40]

Example 6.1 (Partially Ordered Set). If we choose only rule 1, our subjects then form a partially ordered set. Any subject may access any comparable and lesser subject.

An algorithm for deciding the general safety problem may be constructed to execute in linear time: construct the closure of the transitivity relation; the system leaks the right $[s, o]$ if $s \succ o$ in the resultant matrix.

Alternatively, any search treating edges as directed will suffice for an algorithm for the safety problem. If there exists a route from a subject s to an object o then the configuration is unsafe for that subject and object.

Example 6.2 (Set Partition). If we choose rules 1 and 2, we generate a partition of the set of subjects S. Any subject may access any subject within the partition.

An algorithm for deciding the general safety problem may be constructed to execute in linear time: make all edges undirected, construct the closure of the transitivity relation; the system leaks the right $[s, o]$ if $s \succ o$ in the resultant matrix.

Alternatively, any search treating edges as undirected will suffice for an algorithm for the safety problem. If there exists a route from a subject s to an object o then the configuration is unsafe for that subject and object.

Example 6.3 (Undirected Graph). If we choose only rule 2, we generate an undirected graph otherwise identical to the original graph.

An algorithm for deciding the general safety problem may be constructed to execute in linear time: make all edges undirected; the system leaks the right $[s, o]$ if $[s, o]$ in the resultant matrix.

Alternatively, any subject may access any subject to which it is directly connected, if either $[s, o]$ or $[o, s]$ in the initial matrix.

In each case, the alternative algorithm is a special case for the particular protection system and is deduced from knowledge of the construct produced by the particular choice of axioms.

6.4.2 Undecidability of the Class Safety Problem for Boolean Protection Systems

We will demonstrate the undecidability of the safety problem for the boolean access matrix by demonstrating that any protection system with n rights may be reduced to a system with only 1 right, which we will call r.

Theorem 6.8 (Undecidability of the Class Safety Problem for the Boolean Protection System). *It is undecidable whether a given configuration of a given boolean protection system is safe for the right r.*

[40] Briefly, and mostly for our own amusement considering the negative result we are about to roll out.

Proof. Given a protection system $\Psi = (R, C)$, we will construct $\hat{\Psi} = (\hat{R}, \hat{C})$ equivalent to Ψ such that $|\hat{R}| = 1$. Without loss of generality, we let $\hat{R} = \{\hat{r}\}$ and $R = \{r_1, r_2, \ldots\}$. Since there exists a protection system for which the safety problem is undecidable, there must then exist an equivalent boolean system for which the safety problem is also undecidable. It remains only to describe a transformation of an n-right system into a 1-right system which is equivalent on the configuration set of the n-right system.

We give a transformation $\phi : \mathrm{cf}(\Psi) \to \mathrm{cf}(\hat{\Psi})$. This transformation is easier to visualise as a graph, and thus we will describe it using the graph representation suggested in section 4.5.1. The graph representation of $Q \in \mathrm{cf}(\Psi)$ is a structure with directed, discriminated edges. We will produce a structure $\hat{Q} = \phi(Q)$ such that $\hat{Q}$ contains only undiscriminated edges. We may then generate $\hat{C}$ by transforming commands in C to operate on the undiscriminated graph, using a technique similar to that of example 5.2.

The transformation ϕ will replace each directed and discriminated edge $s \xrightarrow{r} o$ with a compound construct of directed but undiscriminated edges such that the construct may be unambiguously identified, using only positive tests, as a representation of the logical link discriminated by r between s and o. There are many possible appropriate transformations, one such transformation is demonstrated in figure 12 on page 92, and described thus:

For a configuration Q, $\hat{Q} = \phi(Q)$ is given by performing the following replacements on every link $r_k \in [s, o]$:

1. The link between s and o is replaced with a chain of links $\hat{r} \in [s, L]$, $\hat{r} \in [L, M]$, $\hat{r} \in [M, N]$, $\hat{r} \in [N, o]$. The reason for the length of this chain is explained below.

2. Two nodes, X and Y, are linked to the midpoint M of the chain, with $\hat{r} \in [X, M]$ and $\hat{r} \in [M, Y]$.

3. A loop of nodes and edges is created from X to Y. The structure representing r_k has a loop of length $2k + 1$. The edges in this loop are in the opposite direction to the edges linking X via M to Y, thus the complete loop $(X, M, Y, \ldots)$ has two reversed edges.

As described in example 5.2 on page 61, we construct commands in $\hat{C}$ which operate on $\hat{Q}$ in the same way that commands in C operated Q. Instead of testing for, granting or revoking rights r into a cell $[s, o]$, any command must test for, construct and destroy compound constructs. An example command transformation for an *enter* operation is also shown in figure 12.

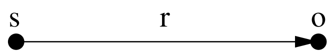
Thus $\Psi \Lleftarrow \hat{\Psi}$ via ϕ.

We must also show that $\Psi \Rrightarrow \hat{\Psi}$ via ϕ^{-1}.[41] It suffices to show that $\phi^{-1}(\hat{Q})$ is well defined, then by a reversal of the transformation applied to commands, $\Psi \Rrightarrow \hat{\Psi}$ via ϕ^{-1}. In turn, it suffices to show that for any $\hat{Q}$, the only subgraphs isomorphic to our chosen discriminated constructs are those which were explicitly generated from some $r_k \in [s, o]$.

Given k, we must find a loop of length $2k + 3$ with exactly two reversed edges. While $\hat{Q}$ may contain many loops, all those formed using only real subjects and objects must contain $4n$

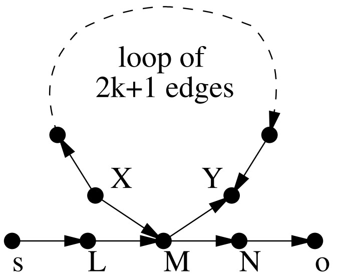
[41] In fact, we mean $\phi^{-1}|_{\phi[\mathrm{cf}(\Psi)]}$, and we should restrict $\hat{Q}$ to be in $\phi[\mathrm{cf}(\Psi)]$. For simplicity, we will simply write ϕ^{-1}. By the niceties of restricted equivalence in section 5.2.4, the result holds.

With n access rights



```
command foo(...)
    if
        ...
    then
        ...
        enter $r$ into $[s, o]$
        ...
end
```

With 1 access right



```
command foo(...)
    if
        ...
    then
        ...
            Create the chain from A to B
        create subject $l$
        create subject $m$
        create subject $n$
        enter $\hat{r}$ into $[a, l]$
        enter $\hat{r}$ into $[l, m]$
        enter $\hat{r}$ into $[m, n]$
        enter $\hat{r}$ into $[n, b]$
            Create X and Y
        create subject $x$
        create subject $y$
        enter $\hat{r}$ into $[x, m]$
        enter $\hat{r}$ into $[m, y]$
            Create and tag the loop nodes
        create subject $s_0$
        enter $\hat{r}$ into $[s_0, s_0]$
        ...
        create subject $s_{k_r+1}$
            Create the loop links
        enter $\hat{r}$ into $[m, s_0]$
        enter $\hat{r}$ into $[s_0, s_1]$
        ...
        enter $\hat{r}$ into $[s_{k_r}, s_{k_r+1}]$
        enter $\hat{r}$ into $[s_{k_r+1}, m]$
        ...
end
```

Figure 12: The transformation of a privilege into a boolean construct

edges in each direction, for some $n > 1$. Therefore, in order to form a loop of length $2k+3$, we must include some of our deliberate loops. We can include at most one of these, or we would have too many reversals. $n > 1$, therefore $4n > 4$, therefore the two reversed edges must be supplied by one of our deliberate loops. But there is no k such that $2k+1 = 2$ to provide the two reversed edges. Thus the loops from X to Y are unambiguous. Since there are two reversed edges, M is unambiguous. Since there are two continuous (nonreversing) edges two from M to s and M to o, s and o are distinct from any nodes in the loop. Thus the construct is unambiguous for all k, and $\phi^{-1}(\hat{Q})$ is well defined.

Thus $\Psi \Rrightarrow \hat{\Psi}$ via ϕ^{-1} and $\Psi \equiv \hat{\Psi}$.

□

Corollary 6.2 (Polynomial Size of Boolean Protection Systems). *Let Ψ be a protection system, and $\hat{\Psi}$ a boolean protection system such that $\Psi \equiv \hat{\Psi}$ via ϕ. If $Q \in \mathrm{cf}(\Psi)$ and $\hat{Q} = \phi(Q)$, then $|\hat{Q}|$ is polynomial in $|Q|$.*

Proof. Let $\Psi = (R, C)$, $Q = (S, O, P)$ and $\hat{Q} = (\hat{S}, \hat{O}, \hat{P})$. The number of rights in P is bounded above by $|S||O||R| \overset{\text{def}}{=} |Q||R|$. Each of these rights introduces at most $2|R| + 5$ new subjects in $\hat{S}$. ϕ introduces no new objects in $\hat{Q}$, thus $\hat{O}$ simply includes the newly created subjects: $\hat{O} = O \cup \hat{S}$. Therefore, we have

$$\begin{aligned}
|\hat{Q}| &\overset{\text{def}}{=} |\hat{S}||\hat{O}| \\
&\leq (|S| + (2|R|+5)|Q||R|)(|O| + (2|R|+5)|Q||R|) \\
&= (|S| + 2|Q||R|^2 + 5|Q||R|)(|O| + 2|Q||R|^2 + 5|Q||R|) \\
&= |Q| + 2|S||Q||R|^2 + 5|S||Q||R| + 2|O||Q||R|^2 + \\
&\quad\; 4|Q|^2|R|^4 + 20|Q|^2|R|^3 + 5|O||Q||R| + 25|Q|^2|R|^2 \\
&\leq |Q| + 10|Q|^2|R| + 29|Q|^2|R|^2 + 20|Q|^2|R|^3 + 4|Q|^2|R|^4
\end{aligned}$$

Which is clearly polynomial in $|Q|$.

□

6.5 Monotonic Protection Systems

An access matrix command may only test for the presence of a right and not for the absence of a right. Therefore any command which may be executed after the removal of some rights, subjects or objects from the access matrix may also be executed if these rights, subjects or objects are still present in the matrix. Therefore, any leaky computation containing **delete** or **destroy** operations is still leaky if the system is made monotonic by the removal of the **delete** and **destroy** operations from each command in the computation.

At first glance, the class safety problem for monotonic systems might appear simpler since we are no longer concerned with the removal of subjects or privileges from the matrix, only their creation within the matrix. However, this is not so and the class safety problem for this class remains undecidable, as shown in [HR78]. We reproduce the proof from that paper in this section.

6.5.1 Construction of the Monotonic Protection System

Definition 6.10 (Monotonic Access Matrix Command). An access matrix command is monotonic if the operations op_i are all of the forms

enter r **into** $[s, o]$
create subject s
create object o

Equivalently, it contains no operation of any of the forms

delete r **from** $[s, o]$
destroy subject s
destroy object o

Only constructive operations are allowed within a monotonic command. Data may be created in the matrix but may not be removed. We observe that while any of these creative operations may only create a bounded amount of data, a **destroy subject** or **destroy object** operation may remove a finite but potentially unbounded amount of data from the matrix. The restriction to constructive operations means that each command may modify only a bounded amount of data.

Definition 6.11 (Monotonic Protection System). A protection system is monotonic if all commands within it are monotonic.

[HR78] made the following fuller, but identical, definition.

Definition 6.12 (Monotonic Protection System ([HR78])). A protection system is monotonic if no command contains a primitive operation of the form

delete r **from** $[s, o]$
destroy subject s
destroy object o

6.5.2 Undecidability of the Class Safety Problem for Monotonic Protection Systems

[HR78] proved that the safety problem for monotonic access matrix systems is undecidable by reduction to the Post Correspondence Problem. The notation in this proof is inconsistent with our more modern framework, but since the proof is entirely self-contained, the original notation has been preserved. Some introductory definitions will refresh the memory.

Definition 6.13 (Recursively Enumerable). A language L is recursively enumerable if a Turing machine with w on the tape will terminate with $\mathbf{Y}$ on the tape if $w \in L$ and will hang or not terminate if $w \notin L$.

Definition 6.14 (Recursive). A language L is recursive if a Turing machine with w on the tape will terminate with **Y** on the tape if $w \in L$ and will terminate with **N** on the tape if $w \notin L$.

Definition 6.15 (Post Correspondence Problem ([Pos46])). Given an alphabet Σ, an instance of Post's Correspondence Problem of size s is a finite set of pairs of strings $(x_i, y_i), (i = 1 \ldots s, s \geq 1)$ over the alphabet Σ. A solution of length $n \geq 1$ to this instance is a sequence $i_1, i_2, \ldots, i_n$ of selections such that the strings $x_{i_1} x_{i_2} \ldots x_{i_n}$ and $y_{i_1} y_{i_2} \ldots y_{i_n}$ formed by concatenation are identical (from [Min72]).

The Post Correspondence Problem is clearly recursively enumerable but is not recursive (for sizes at least 7, [MS96]). Thus it is possible eventually to find the solution to an instance of the Post Correspondence Problem if such a solution exists, but not to state that a solution does not exist in the case that no solution exists. This corresponds with our construction of the general safety problem in section 4.4 in that it is possible to prove any protection system unsafe by finding a leaky computation but it is not always possible to show that a protection system is safe by proving that there is no leaky computation. This property is used by [HR78] in the following proof.

Theorem 6.9 (Undecidability of the Class Safety Problem for Monotonic Protection Systems ([HR78])). *It is undecidable whether a given configuration of a given monotonic protection system is safe for a given right.*

Proof. We will encode an instance of the Post Correspondence Problem ([Pos46]) into the access matrix. The safety problem will be shown to be equivalent to the decision problem for the Post Correspondence Problem.

We would like to be able to grow an x-list and a y-list and at a suitable point in time, to compare them. Because of the monotonic restriction, the x and y lists must be 'interlaced' and the check for equality is done by 'pointer-chasing'.

Formally, suppose we have an instance of the Post Correspondence Problem given by

$$x = (x_1, x_2, \ldots, x_n)$$

$$y = (y_1, y_2, \ldots, y_n)$$

where $x_i, y_i \in \{0, 1\}^+$. It is convenient to define

$$x_i = x_{i1} \ldots x_{il_i}$$

$$y_i = y_{i1} \ldots y_{im_i}$$

where $x_{ij}, y_{ik} \in \{0, 1\} \forall i, j, k$ such that $1 \leq i \leq n$, $1 \leq j \leq l_i$, and $1 \leq k \leq m_i$.

We shall construct a protection system which has the following set of generic rights

$$R = \{0, 1, \mathbf{link}, \mathbf{start}, \mathbf{match}, \mathbf{yx{-}end}, \mathbf{leak}\}$$

and the following commands: For each i, $1 \leq i \leq n$, we have a procedure

```
command start$_i(X_1, \ldots, X_{l_i}, Y_1, \ldots, Y_{l_i})$
    for $j := 1$ to $l_i$ do create subject $X_j$[42]
    for $j := 1$ to $l_i$ do enter $x_{ij}$ into $[X_j, X_j]$
    for $j := 1$ to $l_i - 1$ do enter link into $[X_j, X_{j+1}]$
    for $j := 1$ to $m_i$ do create subject $Y_j$
    for $j := 1$ to $m_i$ do enter $y_{ij}$ into $[Y_j, Y_j]$
    for $j := 1$ to $m_i - 1$ do enter link into $[Y_j, Y_{j+1}]$
    enter yx-end into $[Y_{m_i}, X_{l_i}]$
    enter match into $[Y_{m_i}, X_{l_i}]$ if $y_{m_i} = x_{l_i}$[43]
    enter start into $[Y_1, X_1]$
end
```

For each i, $1 \leq i \leq n$, we have a procedure

```
command grow$_i(X_{\text{end}}, Y_{\text{end}}, X_1, \ldots, X_{l_i}, Y_1, \ldots, Y_{l_i})$
    if
        yx-end $\in [Y_{\text{end}}, X_{\text{end}}]$
    then
        for $j := 1$ to $l_i$ do create subject $X_j$
        for $j := 1$ to $l_i$ do enter $x_{ij}$ into $[X_j, X_j]$
        for $j := 1$ to $l_i - 1$ do enter link into $[X_j, X_{j+1}]$
        for $j := 1$ to $m_i$ do create subject $Y_j$
        for $j := 1$ to $m_i$ do enter $y_{ij}$ into $[Y_j, Y_j]$
        for $j := 1$ to $m_i - 1$ do enter link into $[Y_j, Y_{j+1}]$
        enter yx-end into $[Y_{m_i}, X_{l_i}]$
        enter match into $[Y_{m_i}, X_{l_i}]$ if $y_{m_i} = x_{l_i}$
        enter link into $[X_{\text{end}}, X_1]$
        enter link into $[Y_{\text{end}}, Y_1]$
end
```

For each $b \in \{0, 1\}$, we have a procedure

[42] This is a shorthand for **create subject** X_1 …**create subject** X_{l_i}.

[43] This notation means that this primitive operation is included in the command if $y_{m_i} = x_{l_i}$

```
command match$_b(X_c, Y_c, X_p, Y_p)$
  if
    match $\in [Y_c, X_c]$ and
    link $\in [Y_p, Y_c]$ and
    link $\in [X_p, X_c]$ and
    $b \in [Y_p, Y_p]$ and
    $b \in [X_p, X_p]$
  then
    enter match into $[Y_p, X_p]$
end
```

Lastly,

```
command leak$(X, Y)$
  if
    start $\in [Y, X]$ and
    match $\in [Y, X]$
  then
    enter leak into $[Y, X]$
end
```

Intuitively, this system computes, starting with an empty configuration, as follows: Each command `start`$_i$ encodes strings x_i and y_i into the protection matrix. The location of the first pair of symbols, (x_{i1}, y_{i1}) is marked by **start** while the last pair, (x_{il_i}, y_{im_i}) is marked by **yx-end**.

Each command `grow`$_i$ adds x_i and y_i to the end of some sequence of xs and ys which have been previously entered into the matrix. The locations of the ends of such a sequence are indicated by the **yx-end** right. Similarly, `grow`$_i$ marks the end of the new sequence with **yx-end**.

Notice that `grow`$_i$ is conditional only upon some **yx-end**, which is never deleted. Thus, several different `grow`$_i$ commands may be applied to the same **yx-end**. Each `grow`$_i$ may then be thought of as growing a new branch on each of two trees — one in which paths from the root represent sequences of xs, the other representing corresponding sequences of ys. The **start** right associates the roots of the two trees, while the **link** rights associate ancestors and descendants, and finally the **yx-end** rights indicate ends of corresponding paths. Moreover, the `start`$_i$ commands are unconditional so that we may actually get a forest of these pairs of trees.

Before starting the formal proof, an intuitive example will be worked. Suppose

$$x = (01, 1)$$

$$y = (0, 11)$$

	X_1	X_2	Y_1	X_3	X_4	Y_2	X_5	Y_3	Y_4
X_1	0	**link**							
X_2		1		**link**			**link**		
Y_1	**start** **match** **leak**	**yx-end**	0			**link**		**link**	
X_3				0	**link**				
X_4					1				
Y_2					**yx-end**	0			
X_5							1		
Y_3		**match**						1	**link**
Y_4							**yx-end**		1

Figure 13: The Post Correspondence Problem in a monotonic protection system

Imagine that the following sequence of commands is executed:

$\text{start}_1(X_1, X_2, Y_1)$
$\text{grow}_1(X_2, Y_1, X_3, X_4, Y_2)$
$\text{grow}_2(X_2, Y_1, X_5, Y_3, Y_4)$
$\text{match}_1(Y_4, X_5, Y_3, X_2)$
$\text{match}_0(Y_3, X_2, Y_1, X_1)$
$\text{leak}(Y_1, X_1)$

Figure 13 on page 98 displays the matrix after this sequence has been executed.

We attempt to match corresponding x and y sequences by working from the leaves at the bottom of the tree to the root at the top. This seems easier than working down from the root, since there is a unique chain of links to follow from any node to the root in each tree, whereas working down from the root, it is not clear how to arrange to follow corresponding paths through two trees. The `start`$_i$ and `grow`$_i$ commands start matching two corresponding sequences by matching their last symbols. The `match`$_b$ commands then compare the two predecessors (i.e. ancestors in the tree) of any pair of matched nodes.

The **leak** right can be entered if and only if matching proceeds all the way up to the root nodes. Next, we show that this can happen if and only if the Post Correspondence Problem has a solution; this is known to be a recursively unsolvable problem ([Pos46]). Thus, we will have shown that it is recursively unsolvable whether or not this protection system is safe for the right **leak** and the empty initial configuration.

Notation. Let Φ be the empty configuration $(\emptyset, \emptyset, \emptyset)$. For any configuration (S, O, P) and any $X \in S$, let $A(X) = \{Y \in S \mid \textbf{link} \in P[Y, X]\}$. (In our tree representation, $A(X)$ is the parent, or ancestor, of node (X, X).) We may extend this notation by writing $A^{i+1}(X) = A(A^i(X))$. Let $C(X)$ be the contents of $P[X, X]$.

Lemma 6.1. *If $\Phi \vdash^* Q$ where $Q = (S, O, P)$, then for all $X \in S$, we have*

1. $P[X, X] = \{0\}$ *or* $\{1\}$
2. $|A(X)| \leq 1$
3. $A(X) = \emptyset$ *if and only if there is a* Y *such that* start $\in P[X, Y]$ *or* start $\in P[Y, X]$. *Furthermore, any such* Y *is unique.*
4. *For each* $Y \in S$, *if yx-end* $\in P[Y, X]$ *then there exist* $m \geq 1$, $i_1, \ldots, i_m$ *each between* 1 *and* n *such that*[44]

$$x = x_{i_1} \ldots x_{i_m} = C(A^{\lg(x)-1}(X)) \ldots C(A(X))C(X)$$

$$y = y_{i_1} \ldots y_{i_m} = C(A^{\lg(y)-1}(Y)) \ldots C(A(Y))C(Y)$$

$$\text{start} \in P[A^{\lg(y)-1}(Y), A^{\lg(x)-1}(X)]$$

5. *For each* $Y \in S$, *if* match $\in P[Y, X]$ *then there exists* $m \geq 1$, $X', Y' \in S$ *such that*

$$\textit{yx-end} \in P[Y', X']$$

$$Y = A^{m-1}(Y')$$

$$X = A^{m-1}(X')$$

and for each $j, 0 \leq j < m$, *we have*

$$C(A^j(X')) = C(A^j(Y'))$$

This claim formalises the discussion above. In (1), it is shown how strings are encoded on the diagonal. In parts (2) and (3) every node has a unique parent except the root. In part (3), the root and only the root of every tree is paired with some other tree, and that tree is uniquely determined. **yx-end** joins the ends of corresponding sequences in paired trees according to part (4). Finally, in (5), matching proceeds along corresponding sequences.

We are now ready to do the proof.

The argument is an induction on the length of the computation of $\Phi \vdash^* Q$.

Basis: The argument is trivial for $\Phi \vdash^0 \Phi$.

Induction Step: Assume that $Q = (S, O, P)$ satisfies the conditions. We will show that $Q \vdash_\alpha Q'$ where $Q' = (S', O', P')$ implies that Q' does also.

[44] There is a natural identification taking place here. If we concatenate the contents of the appropriate cells of the matrix, these lines each become something like $x = x_1 x_2 = 011 =$ "0" "1" "1".

If $\alpha = \texttt{start}_i$, it is clear that all the rights entered are placed into created entries so that the 'old portion'[45] of P' is unchanged. Moreover, the "new portion" of P' satisfies conditions (1) through (5) by construction. There is no possible connection between the old and new portions of P' because

$$P'[X, X'] = P[X', X] = \emptyset$$

with $X \in S$, $X' \in S' - S$. Thus $\vdash_\alpha$ where $\alpha = \texttt{start}_i$ preserves (1) through (5).

If $\alpha = \texttt{grow}_i$, it is clear that one of 0 and 1 is entered to each new diagonal element, and all other entries are made off the diagonal, so condition (1), $P[X, X] = \{0\}$ or $\{1\}$ will still hold. The **link** right is never entered in an old object, and only entered once in each new object, so condition (2) ($|A(x)| \leq 1$) still holds. The **start** right is not entered so (3) still holds. If yx-end $\in P'[Y, X]$ with $Y, X \in S$ (not $S' - S$), then (4) holds with x_i and y_i continuing the sequence ending at $(Y_{\text{end}}, X_{\text{end}})$. That is, we have yx-end $\in P[Y_{\text{end}}, X_{\text{end}}]$ and there exist $m \geq 1, i_1, \ldots, i_m$ such that

1. $x = x_{i_1} \ldots x_{i_m} = C(A^{\lg(x)-1}(X_{\text{end}})) \ldots C(X_{\text{end}})$
2. $y = y_{i_1} \ldots y_{i_m} = C(A^{\lg(y)-1}(Y_{\text{end}})) \ldots C(Y_{\text{end}})$
3. $\text{start} \in P[A^{\lg(y)-1}(Y_{\text{end}}), A^{\lg(x)-1}(X_{\text{end}})]$

Finally, (5) is not affected at all in the old portion of P' and holds vacuously in the new portion of P' except possibly for $P[Y_{m_i}, X_{l_i}]$ in the case where $y_{m_i} = x_{l_i}$. In that case, it holds with $m = 1$. Thus, $\vdash_\alpha$ where $\alpha = \texttt{grow}_i$ preserves (1) through (5).

If $\alpha = \texttt{match}_i$, we see that the link right is not entered anywhere so conditions (2) through (4) are not affected. The other rights are entered by this command off the main diagonal, so property (1) is also unaffected. If we can execute

$$\text{match}_b(Y, X, Y_p, X_p)$$

then we must have had match $\in P[Y, X]$. Then, by property (5), there must have been $m \geq 1$, $X', Y' \in S$ such that

$$\text{yx-end} \in P[Y', X']$$

$$Y = A^{m-1}(Y'), \qquad X = A^{m-1}(X')$$

$$C(A^j(X')) = C(A^j(Y'))$$

for each $j, 0 \leq j < m$. It is clear that after the $\texttt{match}_b$ command is executed, similar conditions hold in P' since

$$Y_p = A(Y), \qquad X_p = A(X), \qquad C(Y_p) = C(X_p)$$

and the other entries are unchanged, so (5) is satisfied by $m + 1$, X' and Y'.

[45] We prefer to say 'old portion' of P' rather than $P' \cap S \times O \times 2^R$ and 'new portion' of P' rather than $P' \cap (S' - S) \times (O' - O) \times 2^R$.

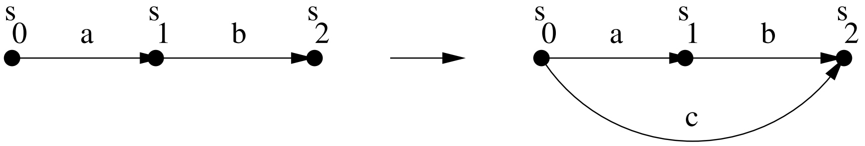


Figure 14: A Rule in a Grammatical Protection System

If $\alpha =$ `leak`, then no rights are entered on the diagonal, so property (1) still holds. **link**, **start** and **match** are not entered, so properties (2)-(5) are unaffected.

Thus, the induction is extended, and we see that lemma 6.1 is true.

We are now ready to prove theorem 6.9 from page 95. Suppose that the Post Correspondence Problem (from definition 6.15 on page 95) has a solution, say $(i_1, i_2, \ldots, i_m)$. Then commands $\texttt{start}_{i_1}, \texttt{grow}_{i_2}, \ldots, \texttt{grow}_{i_m}$ could be executed with appropriate parameters so that the indicated solution is constructed. Since a solution ends with i_m, we certainly must have the "*enter match …*" command in "$\texttt{grow}_{i_m}$", so execution of several "$\texttt{match}_b$" commands with appropriate parameters would result in placing the **match** right in the same position as the **start** right, this allowing the `leak` command to enter the **leak** right. Conversely, if **leak** is ever entered, it must be because **start** and **match** appear in the same position of the matrix. By property (5) of lemma 6.1, we see that there must be some Y', X' such that yx-end $\in P[Y', X']$, and their predecessors match. But then by property (4) for Y', X', we see that their predecessors must be corresponding sequences of x_i's and y_i's, that is, the Post Correspondence Problem must have a solution. Thus, the protection system is safe for the right **leak** and the initial configuration Φ if and only if the Post Correspondence Problem has no solution, and hence safety is recursively unsolvable.

□

6.6 Grammatical Protection Systems

Grammatical systems are an extremely large class of systems, which include the system first described in example 4.1 on page 39, the full Take-Grant model (in [JLS76]), some Role Based Access Control models (in section 7.8, [SCFY96] and elsewhere) and many of our practical protection systems from section 7.

Grammatical systems originated in early early work by Anita Jones, Richard Lipton and Timothy Budd. The concepts start to appear as early as [JLS76] and [LS77], they were formally introduced in [LB78], and study culminated with the positive results in [Bud83]. We will illustrate grammatical protection systems by reproducing many of the ideas from that paper here.

6.6.1 Construction of Grammatical Protection Systems

A grammatical protection system is a system within which all commands may be described as "*If* $r \in [s, x]$ *and* $r' \in [x, o]$ *then we may grant* $r'' \in [s, o]$." Such a rule is illustrated in figure 14. Any such rule may be associated with a set of productions in a context-free grammar. Since the diagram of figure 14 can be reflected, we will have two productions for each rewriting rule, where the second is an inversion of the first. Our rule of figure 14 would be associated with the following two productions:

$$\begin{array}{ccc} \overrightarrow{c} & \rightarrow & \overrightarrow{a}\,\overrightarrow{b} \\ \overleftarrow{c} & \rightarrow & \overleftarrow{b}\,\overleftarrow{a} \end{array}$$

Given a grammar $\mathcal{L}$ of such rules, the language $\mathcal{L}(a)$ is the set of words that can be derived from the initial symbol a. The more formal definition of a grammatical system, which was given as the first lemma of [LB78] and appears as the first theorem of [Bud83], is as follows:

Definition 6.16 (Grammatical Protection System). A grammatical protection system is a protection system such that there is a grammar $\mathcal{L}$ such that for each $r \in R$, subject s and object o, we may grant $r \in [s, o]$ if and only if s and o are connected by a path such that the concatenation of the right symbols on that path form a word in $\mathcal{L}(r)$.

6.6.2 Decidability of the Class Safety Problem for Grammatical Protection Systems

Theorem 6.10 (Decidability of the Class Safety Problem for Grammatical Protection Systems). *It is decidable whether a given configuration of a given grammatical protection system is safe for a given right.*

The reader may refer to [Bud83] for the full proof, which we sketch here.

Proof. [Bud83] observes that grammatical protection systems have the '*Church-Rosser*' property: Given an initial configuration G to which many transformations may be applied, we will achieve the same resulting graph regardless of the order in which these transformations are applied. It therefore makes sense to refer to the graph $\mathcal{G}$, which is the union of all graphs which can be produced by some sequence of transformations starting from G. This $\mathcal{G}$ therefore contains every right which may be leaked from the starting configuration G.

[Bud83] produces a procedure for computing $\mathcal{G}$ from any G in $O(|R||E'||V|P)$ time and $O(|R||E'|)$ space where $G = (V, E)$, $\mathcal{G} = (V, E')$, and P is the number of productions in the grammar.

Thus the safety problem is decidable for grammatical protection systems.

□

It follows from the derivation of a grammatical protection system that any such system is monotonic. If this condition is removed, then [Bud83] shows that the class safety problem for nonmonotonic not-quite-grammatical protection systems is complete on polynomial space.

The assumption of the existence of the maximal graph $\mathcal{G}$ also precludes the use of the *create subject* operation, thus reducing the presentation of grammatical protection systems in [Bud83] to a special case of a bounded protection system, which we described in section 6.1.

6.7 Summary

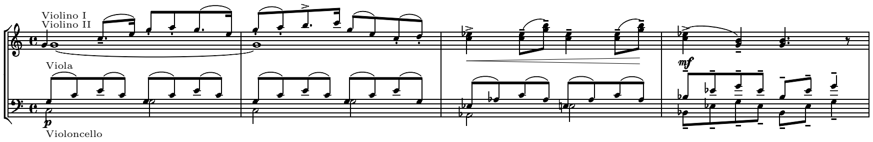
There are many classes of protection system which have either appeared in the literature, or appear likely candidates for establishing a useful result. The selection here includes only a sample of the earlier work and a few likely new candidates. It is intended to cover the simplest and most common classes of system. The extremely general nature of these classes creates much scope for negative results which, while of some academic value, are of little use in the real world. Therefore we will summarise at this point, and move on to the study of some truly *practical* protection systems for which positive results are available.

- Systems for which the class safety problem is not decidable:
  - Boolean systems
  - Monotonic systems
- Systems for which the class safety problem is decidable but has high algorithmic complexity:
  - Object bounded systems
  - Subject bounded systems
  - Monooperational systems
- Systems for which the class safety problem is decidable and has low algorithmic complexity:
  - Monoconditional systems
  - Grammatical systems

Material which follows from this section may be found in:

- Section 7: Our practical protection systems are monooperational. However, the result from this section is not computationally useful, and we produce a better result for our practical systems.
- Section 9: Many of the classes in this section were chosen for their relevance to real world systems. Unfortunately, most real systems bias towards paranoia in the security-flexibility trade-off, and so when the transport of rights is allowed, it is usually one of the simpler systems.
- Section 12: Some more classes which may be worth of study are suggested at the end of this thesis. We have preliminary conjectures about the decidability of some of these classes, but as yet, no proofs have been constructed.

Got anything more useful . . . ?



7 Practical Protection Systems

7.1 Introduction

Decidability of the safety problem is a laudable theoretical goal, but in order to make practical use of a protection system, it really needs to be decidable in low order polynomial time. In this section, we identify and study some examples of protection systems for which the safety problem is decidable in polynomial time. In fact, each and every one of these systems has a class safety problem which is decidable and safety is computable in time linear in the size of the configuration.

However, the decidability of safety is not the only important goal towards which we strive in the design of a truly practical protection system. In order to be practical, a protection system must satisfy many other requirements, some of which were introduced in section 2.4.3. The requirements relating to the safety problem and the expressiveness of the system were not previously suggested in section 2.4.3, since we did not then have the language to discuss them.

- Low computational cost for the system.
- Low computational cost for the system safety problem.
- Expressiveness of available configurations.
- Administrative flexibility and convenience.
- Secure delegation of security system administration.
- Human-oriented justification of all access decisions.
- Availability of accounting and auditing information.

We do indeed design '*ideal systems*' matching all of these requirements, the first of which is described in section 7.10. We discover that system at the culmination of a journey through a world of simpler systems during which we will meet many old friends, including IRC, SQL, Multics and RBAC.

7.2 Designing for Simplicity

A protection system exists for which safety is decidable but arbitrarily difficult, by corollary 4.2 to theorem 4.1. Thus, even if we restrict all of our systems to fall within a class from section 6 with a decidable class safety problem, this will not guarantee that the class safety problem for the system will be decidable in any reasonable time.

Following this, in section 4.5.3, we noted that it is possible to express an arbitrary formula in *'positive atom'* boolean logic using the guard clause of a single access matrix command. If we view the access matrix as the adjacency matrix of a directed graph with labelled edges, then this statement is equivalent to saying that it is possible to require an arbitrary subgraph to be present in the graph of rights. We could, for example, construct an access matrix command **testclique**$(X_0, X_1, \ldots, X_k)$, the guard clause of which tests for the presence of a k-clique in the matrix. Testing for the applicability of just this one command in practice is NP-Hard, thus any practical algorithm to test the safety problem for a protection system containing this command will be at least NP-Hard, if decidable at all.

For the practical purposes of actually constructing a protection system, we will wish to restrict our attention to those systems for which the safety problem is decidable in low order polynomial time. As with section 6, we must focus our attention on particular systems in the hope of discovering a suitable system. However, this time we will build from the bottom up using business requirements, instead of restricting from the top down using the formalisms.

We will start by focusing on the simplest possible system and build upon it with small changes until we have created a system which meets our requirements.

7.3 A Hierarchy of Models

As we explore and combine design decisions to build a system, we will naturally construct a hierarchy or space of protection systems. Each differs from any neighbour by only one design decision, but the impact of each decision may be considerable. In figure 15 on page 106, we display fragments of example configurations of each of a sub-hierarchy of simple protection systems; each figure is centred around the relationships of one principal, object or role. For each model, we will describe a mechanism for permitting access to objects and a mechanism for modifying the protection mechanism within itself. It will become clear as we explore some of the models that many of the design decisions are orthogonal and that figure 15 actually shows a slightly contorted slice through a larger space of models, the dimensions of which are defined by these possible design decisions. The slice which we have chosen includes some common models from the real world, and our selection should give an understanding without explicit study of what lies outside the slice.

As we walk around this diagram, we will learn that there are three clear models for controlling access to objects: "*primitive*", "*role based*" and "*transitive*". With each of these models, we may use one of a number of models for controlling the modification of rights within the

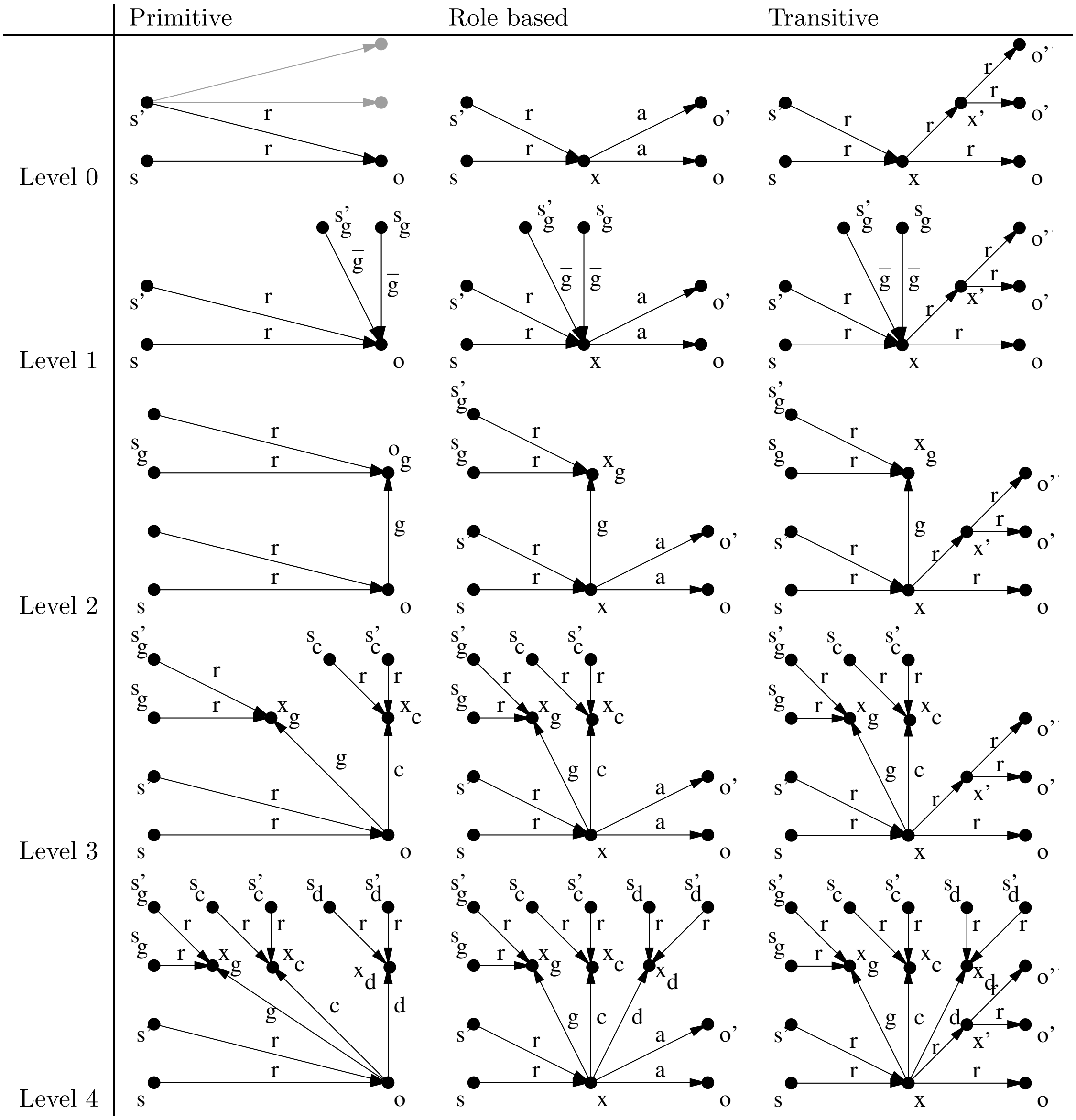


Figure 15: The Increasingly Enraged Hedgehog

system. We indicate these by introducing slightly artificial '*levels*', presenting more complex models at higher levels.

In order to build a flawless and capable system, the mechanisms for the control of access to objects and for the modification of rights must cooperate and may not be selected entirely independently. This is not a cause for compromise since there are clear arguments for choosing particular partnerships. We will gradually unify these two mechanisms in order that we may eventually consider the configuration of the access matrix itself as an object, as suggested in section 2.1.4.

All of the models under consideration are monooperational. In the proof in section 6.3.2, we noted that any leaky computation in a monooperational system remains leaky if all the commands the body of which contains a **delete** or **destroy** operation are removed from the computation. Therefore we will consider only the monotonic variant of each model without loss of generality with regard to the safety problem for any model.

7.4 Level 0 Primitive

7.4.1 Formalisation

This is the simplest system. Access is controlled directly by the presence or absence of the right r.

```
command primitive_access($O$)
    let $s$ = current_principal
    if
        $r \in [s, O]$
    then
        allow_access
end
```

Permission to grant r to an object is controlled by the same access right r.

```
command primitive_0_grant($O, T$)
    let $s$ = current_principal
    if
        $r \in [s, O]$
    then
        enter $r$ into $[T, O]$
end
```

If you have access to an object then you can grant access to any principals, who can then in turn grant access to that object to any further principals. Most simple key-based systems are of this form: possession of the key allows both access and copying of the key.

7.4.2 The Class Safety Problem

A configuration is unsafe for $r \in [., o]$ if there is an untrusted principal s such that $r \in [s, o]$. The configuration is also unsafe if there is an untrusted principal s' such that s' may grant $r \in [s, o]$ to some untrusted principal s, in other words, if $r \in [s', o]$: but this is the same as the previous case.

It is possible to identify whether there is an untrusted principal s such that $r \in [s, o]$ in time linear in the number of subjects. Therefore the class safety problem is decidable for level 0 primitive systems.

7.4.3 Examples

This mechanism is used by many trivial systems such as personal information managers and simple web sites. Notably, this mechanism is used by IRC servers to control access to channel operator commands ([NWG93]). Any channel operator may use the channel operator commands. He may also promote any other user to channel operator status, and that user may then promote further users. As any seasoned IRC user will recognise, the model fails when two operators disagree about who should have which rights. The ensuing confrontation usually involves a large number of rapid promotions and demotions and frequent exploitations of race conditions within the mechanisms of the IRC network. In recent years, this issue has been addressed by many IRC networks by moving to a level 1 model similar to that in section 7.6.

The illustration of level 0 primitive in figure 15 includes two principals and three objects. In general, any one of our illustrations is supposed to be a fragment of a larger configuration, usually focused around the central element of the model (an object, a role, a hierarchy). Some additional objects have been included (in grey) in this first illustration to indicate that this, and all diagrams, are incomplete fragments drawn to include at least one and frequently only one of each major entity in the system.

7.4.4 Summary

Level 0 systems are the simplest possible systems. Any principal who holds a privilege may use it or delegate it freely without restriction. This lack of restriction on delegation is rarely, if ever, desirable, and all following systems attempt to introduce some control over the redelegation of permissions.

7.5 The Problem of Vetting

There is a significant problem with the lack of separation in the level 0 models between the possession of a right and the ability to pass on that right. Conventionally, we accept that such a separation must exist, but we now attempt to justify it in more logical terms.

We grant a right only if we have verified that the target principal may be trusted to act '*responsibly*' with that right (we have '*vetted*' the principal for acting with the right). In the case of granting a simple right r permitting access to an object and nothing else, we must verify

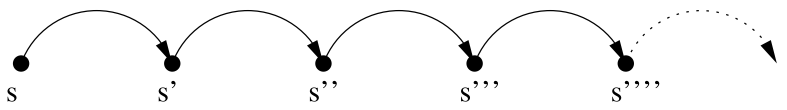


Figure 16: The transitivity of the grant operation: ss grants to s' who grants to s'' ...

that the target principal may be trusted not to perform any malicious operation on the object. However if r also permits itself to be copied to further principals, then we must also verify that the target principal is capable of performing a verification of trustworthiness in a further principal. We now refer to figure 16 and take a big breath because the problem is actually worse than this: even if s can verify that s' can verify that s'' is trustworthy to act with the right r, can s verify that s' can verify that s'' can verify that a new principal s''' is trustworthy to act with the right r. By induction, an arbitrary level of verification is required.

Thus we must discard the level 0 systems and any system in which the possession of right gives a principal the ability to grant that right. Level 1 systems suffer this problem, but we describe them since they are an important stepping stone to the level 2 systems.

7.6 Level 1 Primitive

7.6.1 Formalisation

Access is again controlled by the presence or absence of the right r, as in command **primitive_access** in section 7.4. However, we have a very slightly different mechanism for granting rights. We introduce a new right $\bar{g}$, the possession of which confers the right to grant r.

```
command primitive_1_grant_r($O, T$)
    let $s$ = current_principal
    if
        $\bar{g} \in [s, O]$
    then
        enter $r$ into $[T, O]$
end
```

And of course we must choose a mechanism for granting $\bar{g}$. We again use the simplest possible mechanism, borrowed directly from the level 0 systems.

```
command primitive_1_grant_ḡ($O, T$)
    let $s$ = current_principal
    if
        $\bar{g} \in [s, O]$
    then
        enter $\bar{g}$ into $[T, O]$
end
```

Possession of $\bar{g}$ gives a principal the right to grant r to itself, so most implementations will automatically grant $r \in [s, o]$ to any principal s such that $\bar{g} \in [s, o]$ or allow access using the right $\bar{g}$ as well as the right r. $\bar{g}$ is in some sense a '*greater*' right than r (in a sense related to the calculus of safety from section 5.4), although we will not need to develop this idea as we will instead develop better systems which do not use this type of architecture.

7.6.2 The Class Safety Problem

A configuration is unsafe for $r \in [., o]$ if there is an untrusted principal s such that $r \in [s, o]$. The configuration is also unsafe if there is an untrusted principal s' such that s' may grant $r \in [., o]$, in other words, if $\bar{g} \in [s', o]$. Otherwise, the configuration is safe.

It is possible to identify whether there is an untrusted principal s such that $r \in [s, o]$ or $\bar{g} \in [s, o]$ in time linear in the size of the configuration. Therefore the class safety problem is decidable for level 1 primitive systems.

7.6.3 Examples

ANSI SQL ([fS03], [vdL89], [DD93]) and implementations including MySQL ([RYK02]) and PostgreSQL ([PGDG03]) have a modifier on all access rights called the "`GRANT OPTION`". A principal may hold rights to access data. If they also have the `GRANT OPTION` associated with these rights, then they may pass these rights on to further users, optionally passing on the `GRANT OPTION`. We may consider rights modified by the `GRANT OPTION` as specialisations of $\bar{g}$ for various purposes, and unmodified rights as being similar to r.

DALnet's IRC daemon ([Dal03]) has two levels of channel operator: `SOP` and `AOP`. `SOP`s may grant `SOP` privilege and `AOP` privilege. `AOP`s may grant neither privilege. As before, the `SOP` privilege is $\bar{g}$ and `AOP` is r.

7.6.4 Summary

Level 1 introduces the obvious extension of level 0 systems to include some form of administrative control. However, the problem of controlling this administrative control has not yet been solved, and so in a way, level 1 systems provide an appropriate administrative system for only the most trivial of applications.

7.7 The Problem of Layers

Clearly the choice in section 7.6 of protection for the right $\bar{g}$ will cause our protection system to suffer from the same problems as the level 0 primitive system suffered with the right r, namely that in granting the right $\bar{g}$ to a principal, we are giving the target principal not only the ability to grant r, but the ability to grant $\bar{g}$ to further principals which might not have been vetted as thoroughly as we vetted the target principal. We have reintroduced the problems discussed in section 7.5 at one level distant from the problem of protecting objects.

We can distance ourselves arbitrarily from the problem by introducing instead a new right $\bar{c}$ to control the granting of $\bar{g}$, $\bar{d}$ to control $\bar{c}$, and so on. But we cannot escape the fact that at some fixed and finite point, a right must be responsible for protecting itself. Furthermore, the number of layers is fixed by the number of rights in the system, so in any system implementing multiple layers, it will be common to find that either too few layers or too many layers are present above a particular object.

Our views differ significantly from models presented in [SCFY96] and elsewhere. [SCFY96] specifies a model for RBAC which is protected by a second layer of RBAC, which is in turn protected by a single administrator. He states in defence of this, "*We feel that even a second level of administrative hierarchy is unnecessary. Hence the administration of the administrative hierarchy is left to a single chief security officer.*" This is in direct violation of the axiom of self protection or closure, since now the higher level RBAC system may externally modify the lower level system. Even if we perform a folding of all three layers into the one layer, we still suffer from the "*Problem of Layers*" since the right held by the "chief security officer" is still responsible for protecting itself.

We don't actually manage to address this problem in any significant way until the model of section 7.10, but we have enough other problems to keep us occupied until then.

7.8 Level 1 Role Based

7.8.1 Development

Example 7.1 (The Shared Consultant). Let companies C and D represent the set of their respective employees, who will be the principals of the system. C is an applications company, D is a consultancy company. A consultant $c \in D$ is on permanent attachment to C, thus also $c \in C$ (so $C \cap D$ is nonempty, containing c).

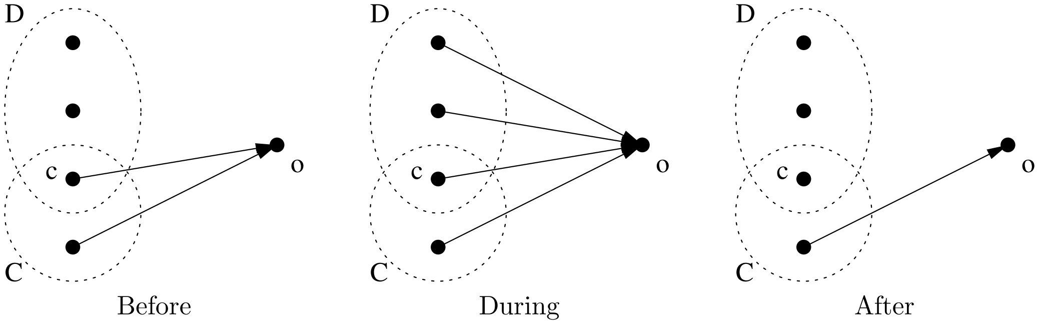


Figure 17: The Shared Consultant without roles

Company C subcontracts company D to work on a project o. Every member of C has access to object o. Each member of D is granted access to the object o. Eventually, the task is finalised and the contract terminates. The administrator of C revokes access to o from every member of D. However, this revokes access to o from c, who is also a member of C and thus should still hold a right to access the object.

The problem here is that the administrator's intention was not to "*revoke o from every member of D*" but to "*revoke o from every member of D who did not otherwise have it for legitimate reasons*". Under a primitive model, each subject has a flat set of rights with no structure indicating *why* a subject had a given right, and possibly also, as in this case, that they had the right for more than one reason.

The role based model introduces a layer of indirection into the access matrix which addresses the problem raised by example 7.1. Roles (x in the "*Role based*" column of figure 15 on page 106) have permissions to access objects and principals hold roles.

Roles may thus be considered as a solution to the problem of example 7.1. Under this mechanism and in the framework of example 7.1, companies C and D would act on the project o in different roles: one as the owner and one as the consultant. The "*During*" phase would then appear as illustrated in figure 18, with "*Before*" and "*After*" being similar but without the role x' and associated arcs. The consultant c would then for the duration of the project be able to access the object o using either of the roles x or x'.

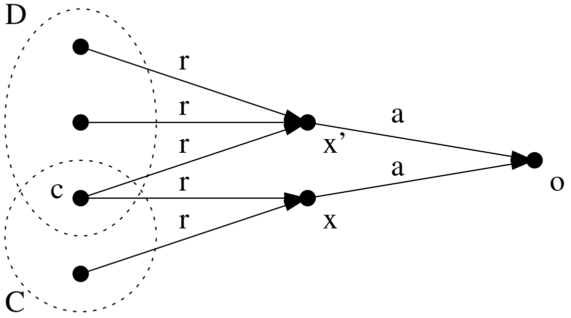


Figure 18: The Shared Consultant with roles ("*During*" only)

7.8.2 Properties of Roles

The introduction of roles gives us the ability to form logical groups of objects and control access to each group as a whole. A role is a collection of privileges; a Christmas box, the possession of which allows a principal the use of a set of privileges to access a set of objects. It is thus that the term "*role*" is justified: A particular set of permissions may be required for performing a task T. This set of permissions may be grouped together and called the *role* x_T for T. While a user may be given the set of permissions required to perform T individually, it is administratively simpler to give the user x_T.

This ease of management makes it easy for a company to remove principals, create new ones and reconfigure the security system appropriately. This can save considerable effort in the case that the relationships between roles and objects do not change as frequently as do the relationships between principals and roles.[46] We are allowed some separation of concern here:

[46] Hypothesis: Programmers get hired and fired faster than jobs get completed?

A project management department may manage the assignment of privileges to roles, while a human resources department manages the assignment of roles to principals.

Roles also allow us to construct some mechanism for justification of access: by inspecting the names of the roles by which a principal may access an object, we may find some human-intelligible justification for that principal to have access to that object. This explanation will be of the form:

> "*Alice is in the UK project group, which uses the Foo code.*"

It is therefore justified that Alice should have access to the Foo code. We will discuss this further in section 7.9.

7.8.3 Formalisation

Principals access objects by choosing a role by which to access the object and using a more complex form of object protection code to access the object through the role. Principals have access to roles using the right r, and objects are assigned to roles using the right a. We note that these commands are no longer monoconditional, and so it is no longer immediate that there exists a fast decision algorithm for the class safety problem for this system.

```
command rolebased_access($O, R$)
    let $s$ = current_principal
    if
        $r \in [s, R]$
        $a \in [R, O]$
    then
        allow_access
end
```

Roles may be granted to principals. The mechanism for controlling this is similar to the mechanism which previously controlled the granting of objects to principals.

```
command rolebased_1_grant_r($R, T$)
    let $s$ = current_principal
    if
        $\bar{g} \in [s, R]$
    then
        enter $r$ into $[T, R]$
end
```

The strengths and weaknesses of this mechanism for controlling the right r were discussed in sections 7.5 and 7.7. We must now have the undesirable command:

```
command rolebased_1_grant_ḡ$(R, T)$
    let $s =$ current_principal
    if
        $\bar{g} \in [s, R]$
    then
        enter $\bar{g}$ into $[T, R]$
end
```

Objects are granted to roles. For purely computational reasons, roles must therefore be subjects of the access matrix. We have defined no mechanism for controlling the granting of objects to roles with the right a. There are many answers to this problem, none of them particularly elegant. Previous works such as [SCFY96] made no specific account of how this relation is to be managed. One simple solution would be to use a new right $\bar{h}$ to control a, with the following command:

```
command rolebased_1_grant_a$(R)$
    let $s =$ current_principal
    if
        $\bar{h} \in [s, o]$
    then
        enter $a$ into $[R, o]$
end
```

Now assuming that the principal s has at least one role x, the privilege $\bar{h}$ is '*greater*' than a in the same sense that $\bar{g}$ was greater than r in the model of section 7.6. s may immediately grant a to that role x and the roles themselves become almost an inconvenience for s accessing o. We must also introduce a mechanism for the control of $\bar{h}$, and we choose the simplest possible mechanism.

```
command rolebased_1_grant_h̄$(T)$
    let $s =$ current_principal
    if
        $\bar{h} \in [s, o]$
    then
        enter $\bar{h}$ into $[T, o]$
end
```

This rapidly gets ugly, and we will emulate many previous works by ignoring the details. These problems are solved by the introduction of transitive systems in section 7.9.

7.8.4 The Class Safety Problem

The safety of an object now relies on there being no role x such that $r \in [s, x]$ and $a \in [x, o]$. In order to prove the system safe, we must show that it is impossible to identify or find such a role. There are now two points of attack.

The computation for the safety of the privilege $r \in [., x]$ is identical to the computation in section 7.4.2 for the safety of $r \in [., o]$. However, in the case of this model, the computation must be repeated once for each role x such that the system is unsafe for $r \in [x, o]$. Both r and $\bar{g}$ must be safe from any untrusted principal.

The computation for the safety of the privilege $a \in [x, o]$ is again similar to that of $r \in [., o]$ in section 7.4.2. In the case that any untrusted principal has at least one role x, if the system is unsafe for $\bar{h} \in [s, o]$, then s may grant $a \in [x, o]$ and access to o has been leaked.

All of these computations may be performed in short time.

7.8.5 Specifics of Implementation

As a rule, the user does not want to be concerned with identifying a suitable role for accessing an object, especially since roles were introduced merely for the convenience of the system administrator who wishes to grant permissions *en bloc.* Solving this dilemma requires a linear time algorithm to identify a suitable role for accessing the object. This algorithm must search over all possible roles until it either finds an appropriate role or fails. A principal may cache this information, but this would require linear storage and the principal would still have to resort to the search algorithm if the administrator changes anything.

7.8.6 Summary

Roles have been discussed extensively in [SCFY96] and come under some practical discussion in [fS03]. The earlier formalisation of the role based mechanisms in [SCFY96] is from an entirely qualitative perspective; that of ease of administration. "*A major purpose of RBAC is to facilitate security administration and review.*" The construction provides only for the convenience of the administrators and suggests no useful method for controlling the modification of the configuration. Our presentation attempts to exhibit the best features of this construction, but we do not claim that it solves any of the problems of protection of the configuration itself: it does not.

Some other points of note:

- Roles represent logical groups of objects.
- The roles by which a principal may access an object represent some justification for that principal to have access to the object.
- In practice, an RBAC system has an inordinately large number of rules.
- Safety is easily decidable, but the algorithm is more complex than necessary.
- Roles are subjects, but the nature of these subjects as principals has not yet been exploited.

7.9 Level 1 Transitive

7.9.1 Development

We have three problems with the role based mechanisms and the level 1 transitive model will solve two of them. They are:

1. The $\bar{g}$ right protects itself and introduces the problems in section 7.5.
2. There is no elegant way of maintaining the a right as described in section 7.8.
3. The shared consultant problem recurs. The structure of roles does not yet match the administrative structure of the real world, as will be explained again in example 7.2.

Of these, we will solve the latter two, and of those, the last still bears some explanation. We also have some remaining unsatisfied qualitative concerns including:

- We can achieve at most two levels of separation of concern.
- We can achieve at most two step justifications of access.
- The configuration of the protection system does not yet mirror any organisational hierarchy.

We can illustrate these problems best with an example.

Example 7.2 (The Shared Project[47]). Let a company have many subsidiaries, say a set C of subsidiaries in England and a set D of subsidiaries in mainland Europe. We will represent each subsidiary as a role, and grant that role to each employee of a company on attachment to the given subsidiary. Any projects on which a subsidiary works will be objects accessible through that role.

Let there be one subsidiary c which exists both in mainland Europe and in the UK (so $C \cap D$ is nonempty, containing c). Let o be a project used in England, so that all roles $x \in C$ have $a \in [x, p]$. Now let o be introduced into mainland Europe so that all roles $x \in D$ have $a \in [x, p]$.

The project is unsuccessful in Europe, so access is revoked from all European roles. However, this removes access to the project from the role c, our friendly trans-national company.

We could reiterate our previous solution from section 7.8 and introduce another layer of indirection. However, this would still be a very limited model and we could easily recreate a similar problem at a greater level of indirection. We must be able to subdivide roles arbitrarily, yet still manipulate them as wholes.

We redefine the relationship between roles as a hierarchy, throughout which we use the same right r. The level 1 transitive example in figure 15 displays two principals s and s' holding a role x which has been partially subdivided into a role x'. Infinitely more complex examples are

[47] Previously known as "*The Shared Consultant*", with one fewer layers of indirection.

possible, and in each case the hierarchy may be chosen to match exactly the logical construction for the system. Previously a role represented a logical group of objects, and was granted to a principal on the basis that the principal fulfilled the task. Now a role represents any logical grouping, possibly containing further roles and objects; the role may either be granted directly to principals or to other roles which represent larger logical groupings.

7.9.2 Properties of Transitive Systems

The administrative tool for justifying access developed in section 7.8 may now be developed fully. Previously, Alice was a member of the UK project group. But why is she so? Alice is a European consultant. Now, by following a path of roles from a principal to an object, we may read a complete explanation of why Alice is in the UK project group:

> “*Alice works for the security subsidiary of a European company which is on secondment to our UK security arm which is a UK project group and must use the Foo code.*”

Each underlined entity in the explanation is a principal (or in the last case, the object), and each relation is an instance of the right r. The transitive nature of access over the right r gives Alice access to the Foo code. This explanation is much more valuable than the one-liner produced by the model of section 7.8 since it can be developed to an arbitrarily level of granularity by an appropriate selection of role hierarchy.

We may now exploit this hierarchy further. If we say,

> “*Bob is the manager of Alice*”

then Bob should have the right to access any object which Alice may access. The simplest and most intuitive way to represent this would be by granting the principal Bob access to the principal Alice, and thus by the transitivity of r, Bob will have access to all objects in the transitive closure of r from Alice.[48] Hence we realise that any principal may be treated as a role, and correspondingly that all roles may be valid principals. The only objects which are not principals are the data objects.

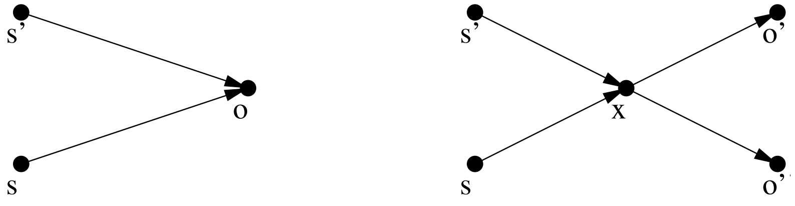


Figure 19: Splitting an object

We can also make a case for data objects to be principals. In figure 19, an object o is accessible by principals s and s'. We want to split o into o' and o''. We also wish to be able

[48] Bob will also have access to Alice, which might or might not be a good thing.

to grant access to o' and o'' separately, yet still refer to o as a single entity for the purpose of protection. We achieve this by constructing a role x, a principal, as in the right hand side of figure 19 and granting $r \in [x, o']$ and $r \in [x, o'']$. However, in order to perform this upgrade smoothly, we must also grant $r \in [s, x]$ for all s such that $r \in [s, o]$. This administrative overhead would be saved if we were able to treat o as a role and merely transfer the data contained in o to objects o' and o''. In this case, x *is* o and no administrative overhead is required to upgrade o to a role.

Any such transformation of objects into principals is a modification of a configuration into a safety equivalent configuration. Therefore, while it makes sense to refer to things accessed as objects and things accessing as principals, the distinction in computational terms between principals and objects is unnecessary. Thus, we may make everything a role, and thus a principal, with no loss of safety and considerable increase of convenience.

7.9.3 Formalisation

In the illustration of the level 1 transitive model in figure 15, both principals s and s' may access any of o, o' and o'' due to the transitive nature of r. Implementations will frequently compute this transitive closure as a graph walk and allow the principal s to access every object reachable over this graph walk.

The result of the graph walk will normally need to be recomputed if the access matrix changes. However, in the introduction to the hierarchy in section 7.3, we stated that we would consider only the monotonic variants of the models, and this allows us to perform this graph walk entirely within the framework of the access matrix by completing the relation r without concern for the problems of revocation. A practical implementation of the nonmonotonic case is discussed in section 9.13.

```
command transitive_infer($S, X, O$)
    let $s$ = current_principal
    if
        $r \in [S, X]$
        $r \in [X, O]$
    then
        enter $r$ into $[S, O]$
end
```

S is not bound to the current principal – indeed, the current principal need not even appear in this command because this is a safety-preserving transformation. Any untrusted principal who might access X would be able to execute this command anyway, were S bound to the current principal. If there is no such untrusted principal then the configuration remains safe. We now have much simpler access code taken directly from the primitive models.

```
command transitive_access($O$)
    let $s$ = current_principal
    if
        $r \in [s, O]$
    then
        allow_access
end
```

We also use the grant command **primitive_1_grant_r** from section 7.6 and control the right $\bar{g}$ using the command **primitive_1_grant_ḡ** from the same section.

7.9.4 Consequences of the Design

In this construction, we remove the following restrictions from the model described in section 7.8:

- That there be at least one layer of indirection between a principal and an object.
- That there be at most one layer of indirection between a principal and an object.
- That a be a right distinct from r and with distinct administrative problems.

7.9.5 The Class Safety Problem

The set A of roles accessible to a principal s includes at least the transitive closure of r from s by the rule **transitive_infer**. But if any $x \in A$ of those roles has a privilege $\bar{g} \in [x, x']$, then by the command **primitive_1_grant_r**, the principal s may grant the privilege $r \in [s, x']$. Therefore the set A is the transitive closure of both r and $\bar{g}$ from s, and the set of unsafe roles (and objects) is given by the transitive closure of r and $\bar{g}$ starting from all untrusted principals.

The command **primitive_1_grant_ḡ** is not relevant to the safety problem in this example. If $\bar{g} \in [s, x]$ for an untrusted principal s, then that principal may acquire $r \in [s, x]$ directly. Granting $\bar{g}$ to another possibly untrusted principal does not create any further leak since every command is monoconditional in $\bar{g}$.[49]

It now becomes startlingly clear that the algorithm for the evaluation of the general safety problem proposed by [HRU76] and described in section 4.6 is incorrect. Were we to remove all trusted principals from the matrix, this would alter the transitive closure of r and would prevent us from finding leaks in the system!

7.9.6 Specifics of Implementation

We do not use the **transitive_infer** command in practice, but rather implement the transitive closure as a graph walk. There are two possibilities for the implementation of this graph walk. The first is to allow a principal access to all objects reachable over 0 or more arcs labelled by

[49] Every command references the right $\bar{g}$ at most once.

r. The second is to allow access to all objects reachable over 1 or more such arcs. The first is probably the most common; the second is formalised above. In order to formalise the first, we must allow the entry of $r \in [s, s]$ for all subjects in the matrix and for any subject s as we create it.

We have not described transitivity of $\bar{g}$. There are several possibilities for identifying whether a principal s may grant a role x including:

- $\bar{g} \in [s, x]$: s has right $\bar{g}$ directly over x.
- $r \in [s, x']$ and $\bar{g} \in [x', x]$: s has access to a role which has right $\bar{g}$ over x.

The first of these options reintroduces the problems of section 7.5, and so we will prefer the second as an introduction to section 7.10, where a better construction is made. In that section, we will see an alternative formulation which does not use $\bar{g}$ as a grant right, but is very similar to the second of these possibilities.

Since $\bar{g} \in [s, x]$ immediately allows $r \in [s, x]$ (as described in section 7.6), the design of this protection system allows us to describe the set of roles accessible to a principal as all those in the transitive closure of r *and* $\bar{g}$ from that principal.

As an unintended but desirable side effect of the transitivity of access, it is now possible to authenticate and authorise each user for exactly one role, and handle all further authorisation by granting roles to that one role. This greatly simplifies the construction of the system since authorisation must be handled at most once for each user, and the set of roles accessible to that user may be easily identified by evaluating a transitive closure from that one role.

Theorem 7.1 (Identity of Users). *A user in a transitive protection system need only be authenticated for one role.*

Proof. If we assume that a user must be authorised for two roles x and x', then we may create a new role x'' and grant $r \in [x'', x]$ and $r \in [x'', x']$. Now we must only authorise the user for x''.

We must show that the new configuration is safety equivalent to the old system. Since we have modified the access specifications of x and x', if the safety of these two nodes is unchanged, then by inspection of the system, the safety of all other nodes is unchanged.

In the new configuration, the node x is unsafe for all principals previously having access to x and all principals having access to x''. This latter includes only the user u, who previously had access to x. Therefore the safety of x is unchanged. A similar argument applies to x'.

Therefore the new configuration is safety equivalent to the old configuration.

□

And thus we justify our limited disregard of the relationship between the user and the principal. We may grant some unique '*user role*' to each user in a system and henceforth identify that user by that role. We then grant all other roles for that user to the user role. Two users with the same user role are indistinguishable to the protection system since the set of roles accessible to each user is the same. This situation, equivalent to the sharing of UIDs or passwords, is usually considered to be a bad idea, since it divorces the model from the real world which it is attempting to represent.

7.9.7 Partial Ordering of Roles

Let the set of all permissions be denoted $\mathbb{P}$ where $\mathbb{P} = 2^{O \times R}$. Then the transitive closure of r and $\bar{g}$ from a role x is some $C(x) \subseteq \mathbb{P}$. Let also the transitive closure of r and $\bar{g}$ from a role x' be $C(x') \subseteq \mathbb{P}$. If $r \in [x', x]$, then $C(x') \supseteq C(x)$, since x is accessible from x' and thus anything accessible from x is also accessible from x'. This naturally encourages our concept of a "*higher*" privilege where "x is higher than x'" and "$r \in [s, x]$ is higher than $r \in [s, x']$", and thus a partial ordering on the set of all roles.

Notation. If a principal or role x has access to a role x' by the transitive closure of the right r, we will write $x \geq x'$.

This partial ordering[50] corresponds with the partial ordering on subsets of all privileges. We use the nonstrict inequality $\geq$ rather than $>$ because if $r \in [x', x]$ but also $r \in [x, x']$ then both $x' \geq x$ and $x \geq x'$, in other words $x = x'$. In this case, $C(x') \supseteq C(x)$ and $C(x) \supseteq C(x')$ so $C(x) = C(x')$ (where $C(x)$ and $C(x')$ are defined as before). From this we see that the set of permissions accessible from x and x' are identical and the roles are indistinguishable for all computational purposes. We may extend this to any ring of roles x_i $(i = 0 \ldots k)$ such that $r \in [x_i, x_{i+1}]$ and $r \in [x_k, x_0]$.

7.9.8 Exploiting and Extending the Transitive Model

Organisational roles naturally fall into hierarchies. At the top of an organisational hierarchy is some object called the 'managing director' or 'board' with total executive power. This object has a role which is superior to, or greater than, all other roles. Immediately inferior to this super-object are departments. Each department head has a role which is greater than every other role in the department.

Consider a sub-hierarchy with one manager T managing developers on two projects X and Y which must talk to one another. Such a situation is illustrated on the left hand side of figure 20. X and Y are incomparable but $T \geq X$ and $T \geq Y$. Each subject is the root of a subtree which itself may be broken down arbitrarily finely such that permissions may be granted to any sub-element of the tree. Conversely, permissions to any entire subtree may be granted if so desired, by granting access to the root of the tree.

This allows for controlling information, but it does not allow for sharing of information between X and Y. What happens when Y wants to verify that a piece of data has come from within the unit defined collectively by X and Y? To what object can X and Y both write in order to share data?

[50] See definition A.2.

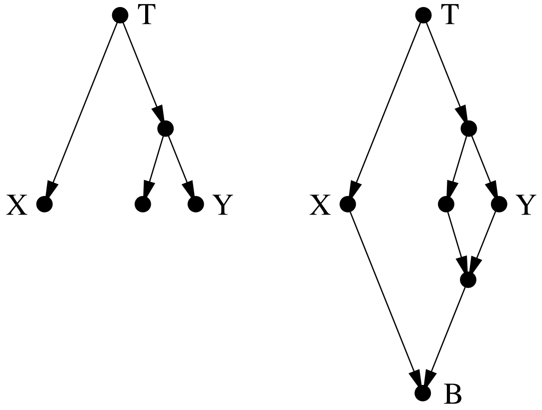


Figure 20: The closure of a tree

The answer is to introduce a closure to this tree. There must exist a role B such that $X \geq B$ and $Y \geq B$. In order to check that this piece of data has come from one of X or Y, a principal must check that the object was sourced by a role superior to B. Shared objects may be protected by role B rather than being granted to both X and Y, and will then immediately become accessible to all present and future sub-roles of T.

The reason for naming this privilege B is that it is the "*bottom*" privilege of the "*top*" privilege T. Usually, nobody is directly granted privilege B; it merely exists in order to allow entities under T to share information or for entities outside of T to verify that information was sourced from some project under the responsibility of T.

We note that the four privileges T, X, Y, B form a diamond with T at the top and B at the bottom. An algorithm to define the bottom privileges given a privilege tree is immediate, and they need not be explicitly represented in the memory of an implementation.

This is the "*diamond pattern*" in transitive protection systems. We exploit this more fully in section 10.1.

7.9.9 Summary

The term "*Role Based Access Control*" is usually used to refer to any one of the '*role based*' models from figure 15. We will use it almost exclusively to refer to the transitive models, since the models which we describe as *role based* are not of much use. Key points from the preceding discussion include the following.

- There is no distinction between principals and roles.
- Every object can be a role, and hence a principal, with no loss of security or generality.
- Each user need be authenticated for at most one role.
- Principals form a hierarchy ordered by the right r.

- Clear and arbitrarily detailed justifications for access are now available.
- The set of roles accessible to any principal is computable in time linear in the size of the configuration.
- We have introduced a notation $x \geq x'$ over roles and principals.
- We have introduced and described the importance of the diamond pattern.
- We may arbitrarily subdivide roles in a hierarchy without computational overhead.

We may now use the words, "*principal*", "*object*" and "*role*" to refer to the same type of entity. The choice of word will serve only for emphasis on how the object is participating in the discussion.

7.10 Level 2 Transitive: The First Ideal System

7.10.1 Development

The remaining unsolved problem from our introduction to the previous section 7.9 is to control the granting of the right $\bar{g}$, as described in section 7.5. If we remove from the protection system all commands referencing r, then we are still left with a level 0 primitive system constructed using $\bar{g}$. This layer was introduced by the choice of some apparently simple grant code for r. However, this design is not mandated, nor is it desirable. We will therefore consider alternative possibilities for commands to grant r without changing the semantics of r. Clearly, any hypothetical grant command for r must grant $r \in [T, O]$ for a specified target principal T and a specified object O.

```
command hypothetical_grant($T, O$)
    let $s$ = current_principal
    if
        $k \in [S, X]$
    then
        enter $r$ into $[T, O]$
end
```

What remains are the choices of S, X and k. For S, we have three possible choices.

- **An arbitrary principal:** a principal chosen by the user at runtime, the rights of whom are used in access control decisions. I would enjoy using `root`'s privileges for my access control decisions, but I would get jumpy were everyone else also to do so. *No.*
- **Any other computed principal:** we may extend the guard clause of the command in order to compute some more relevant principal, but this seems unjustified when we have . . .

- **The current principal:** which has served us well so far, so we will prevail further upon its rights and use it in place of S in our hypothetical command.

And now we must choose X and k. If we ignore the possibility of arbitrary choices, then there are four possibilities.

k X	r	f(r)
O	Level 0 (1)	Level 1[51] (2)
g(O)	Level 2 (4)	[SCFY96] (3)

1. $X = O, k = r$: If we make these substitutions into the hypothetical grant code above, the result is the grant code from the level 0 systems, one of which is described in section 7.4.
2. $X = O, k = f(r)$: $f(r) \neq r$ since this would be identical to the previous case. Therefore $f(r) = \bar{g}$ where $\bar{g}$ is some constant, and we have a level 1 system as described in sections 7.6 to 7.9.
3. $X = g(O), k = f(r)$: This implies that the right controlling $r \in [., o]$ is an entirely separate right on a separate object. This model is proposed by [SCFY96] and its weaknesses were discussed in section 7.8.
4. $X = g(O), k = r$: This is the case which interests us, and which we will call a *Level 2 system*.

Given these choices, our hypothetical_grant command now appears as follows:

```
command reversed_grant_functional($T, O$)
    let $s$ = current_principal
    if
        $r \in [s, g(O)]$
    then
        enter $r$ into $[T, O]$
end
```

This command does not quite fit into the access matrix formalism since it introduces extraneous notation $g(O)$. We can construct an equivalent command within the formalism if we recall that a right may describe any relation and that a single-valued relation is a map.

[51] Actually, we mean Level 1 or higher levels without "*inversion*" of g. Our inversion of g in layer 2 is a deviation from our original progression of introducing further layers of grant rights.

```
command reversed_grant($T, O$)
    let $s$ = current_principal
    if
        $g \in [o, x]$
        $r \in [s, x]$
    then
        enter $r$ into $[T, O]$
end
```

However, this introduces the possibility of there being many such x such that $g \in [o, x]$, whereas previously we referred to g as a map giving $x = g(o)$, in other words, there need exist at most one such x. We therefore defend our reference to g as a map with the following theorem.

Theorem 7.2 (Uniqueness of the Grant Role). *For a given object o, the configuration may be modified into a safety equivalent configuration in which there exists exactly one role x such that $g \in [o, x]$.*

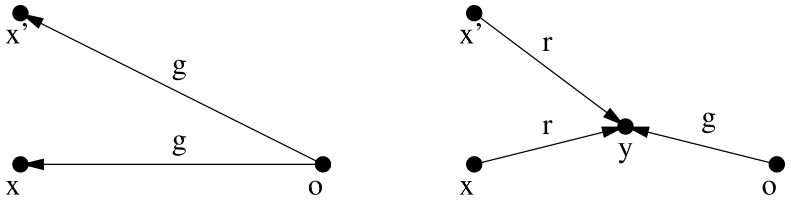


Figure 21: That $\{x \mid g \in [o, x]\}$ may be unique

Proof. Let $g \in [o, x]$ and $g \in [o, x']$, as in the left hand part of figure 21, so x and x' are both grant roles for o. Then we may create a new role y to serve as the unique grant role for o, and grant $r \in [x, y]$ and $r \in [x', y]$. We revoke $g \in [o, x]$ and $g \in [o, x']$ and grant $g \in [o, y]$ to construct the configuration in the right hand side of figure 21.

The new configuration is safety equivalent to the old configuration.

Clearly, the safety of $. \in [s, x]$ and $. \in [s, x']$ is unchanged. We must show that the safety of $r \in [., o]$. But this is unchanged. In the old configuration,

$$\text{safety}(r \in [., o]) = \text{safety}(r \in [., x]) \wedge \text{safety}(r \in [., x'])$$

and in the new configuration,

$$\text{safety}(r \in [., o]) = \text{safety}(r \in [., y]) = \text{safety}(r \in [., x]) \wedge \text{safety}(r \in [., x'])$$

Therefore the new configuration is safety equivalent to the old configuration.

We can repeat this construction for all $\{x \mid g \in [o, x]\}$, therefore for each object o, there exists at most one x such that $g \in [o, x]$.

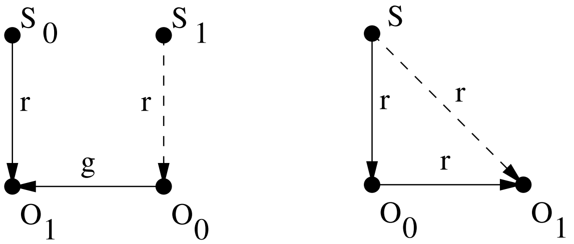
If there is no $\{x \mid g \in [o, x]\}$, then we create a new object y and grant $g \in [o, y]$. Since the system is safe for y, the two configurations are safety equivalent.

□

This construction is similar to that in theorem 7.1 where we demonstrated that each user need be authenticated for exactly one role.

Notation. In a protection system which exploits theorem 7.2, g may be considered as a map and we may write $g(o)$ to mean the principal x such that $g \in [o, x]$.

7.10.2 Formalisation



Granting a right

Inferring a right

Figure 22: The main operations of a level 2 transitive system

The full level 2 transitive system contains the following commands, reproduced from earlier discussion. The operations of the infer and grant commands are illustrated in figure 22.

```
command transitive_access($O$)
    let $s$ = current_principal
    if
        $r \in [s, O]$
    then
        allow_access
end

command transitive_infer($S, X, O$)
    let $s$ = current_principal
    if
        $r \in [S, X]$
        $r \in [X, O]$
    then
        enter $r$ into $[S, O]$
end
```

```
**command** reversed_grant($T, X, O$)
    **let** $s$ = **current_principal**
    **if**
        $r \in [s, X]$
        $g \in [O, X]$
    **then**
        **enter** $r$ **into** $[T, O]$
**end**
```

7.10.3 Consequences of the Design

This system is in many ways similar to one system described in section 7.9.6. In order to grant access to a role x, we must have access to the grant role $g(x)$. However, it differs in that the access required need not be direct, but will usually involve the right r.

In order to grant the ability to grant access to the role $g(x)$, we must have access to any role $g(x')$ where $x' \geq x$. This chain of grant rights will either loop, with $x \geq g(x''''')$, making all roles in the loop equivalent, or terminate in such a loop. The simplest such terminating loop, and the most common, is a single role such that $g \in [\hat{x}, \hat{x}]$. In this case, the role $\hat{x}$ protects itself. This role $\hat{x}$ is then the top of a tree (or diamond) of roles. Layers in roles have not entirely gone away, but have been made sufficiently flexible to reflect the organisational hierarchy being represented. At some point in the organisational hierarchy of vetting[52] for any given right, we either run out of levels in our organisation ('*the board*'), or we run out of significant distinction between the levels, and there we place our $g \in [\hat{x}, \hat{x}]$.

Thus the problem of layers described in section 7.7 has been largely solved by describing the rights $r, \bar{g}, \bar{h}, \bar{c}, \bar{d}, \ldots$ (each controlling the ability to grant the previous) in terms of the right r and inside the matrix itself.

$$
\begin{array}{lll}
s \xrightarrow{\bar{g}} x & \text{is now} & s \xrightarrow{r}^{*} . \xleftarrow{g} x \\
s \xrightarrow{\bar{h}} x & \text{is now} & s \xrightarrow{r}^{*} . \xleftarrow{g} . \xrightarrow{r}^{*} . \xleftarrow{g} x
\end{array}
$$

Where . represents some arbitrary role and $\xrightarrow{r}^{*}$ represents the transitive closure of the right r, zero or more followings of the right r as described in section 7.9.6. We term the transformation of $\bar{g}$ into g an '*inversion*' of $\bar{g}$.

7.10.4 Specifics of Implementation

Most implementations will exploit theorem 7.2 for computational optimisation and administrative simplicity without loss of generality, by making the grant role a function of the role. However, this is by no means required either for implementation of the system or for polynomial time decidability.

[52] See the discussion in section 7.5.

As noted in the specifics of the level 1 transitive model, section 7.9.6 there are again 2 options for the evaluation of the transitive closure of r: following zero or more links, or following one or more links.

Very fast nonmonotonic implementations are possible, using partial results to speed up the graph walk to the point where it may be computed in a very few instructions. The author uses this mechanism in his own systems, one of which is described in section 9.13.

7.10.5 The Class Safety Problem

Theorem 7.3 (Decidability of the Class Safety Problem for Level 2 Transitive Systems). *The class safety problem is decidable for level 2 transitive systems.*

We will prove this by demonstrating an algorithm to decide the class safety problem for transitive systems. We assume that in practice, $r \in [s, s]$, as mentioned in sections 7.9.6 and 7.10.4.

The proof for a given principal and object is remarkably similar to that provided by [JLS76]. However, we can do better than this. We can construct an algorithm to decide the entire set of objects which may be accessed by any untrusted principal! This algorithm may be compared with algorithms for mark and sweep garbage collection such as that in [McC60] or [Jon96].

We assume, under the guidance of section 4.6, that no trusted principal will grant any rights. However, they may still be used, for example as a midpoint of the transitivity rule, as in example 4.3 on page 48. Thus we may not delete trusted principals from the matrix, but we will disallow them from acting as the current principal. By an inspection of the commands available in the transitive system, this means that we may only use the privilege $r \in [s, .]$ when s is untrusted, and we may only use the privilege $g \in [., x]$ when the role x is accessible to an untrusted principal.

Example 7.3 (An Algorithm to Decide the Class Safety Problem for Transitive Systems).

Let (S, O, P) be a configuration of level 2 transitive protection system. Let $U \subseteq S$ be the set of untrusted principals. Let O be the set of all objects (including all roles). Then the algorithm described in algorithm 2 on page 129 will compute the set of all 'unsafe' objects, that is, all objects accessible in eventuality by any untrusted principal.

If an untrusted principal may hold a role x, then all roles for which x is the grant role are unsafe, and all roles held by x are unsafe. Therefore, what we have described above is a mark-sweep algorithm following the right r forwards and g backwards.

While our described algorithm takes at worst $O(n^2)$ time, the garbage collection literature contains techniques which allow us to reduce this to a linear time algorithm ([Jon96]).

Similar algorithms may be constructed to decide the more restrictive RBAC models. These algorithms are specialisations of the solution to the class safety problem for grammatical protection systems described in section 6.6.

Algorithm 2 Algorithm to compute the set of objects accessible by any untrusted principal in a level 2 transitive system

```
Require: A set $U$ of untrusted principals.
Ensure: $U$ will contain all objects accessible by any principal initially in $U$.
  $flag \leftarrow$ true
  while $flag =$ true do
    $flag \leftarrow$ false
    for all $o \in O - U$ do
      for all $u \in U$ do
        if $r \in [u, o]$ then
          $U \leftarrow U \cup \{o\}$
          $flag \leftarrow$ true
        end if
        if $g \in [o, u]$ then
          $U \leftarrow U \cup \{o\}$
          $flag \leftarrow$ true
        end if
      end for
    end for
  end while
```

7.10.6 Summary of the Formalism

The level 2 models bring one new feature onto the scene, the map $g : S \Rightarrow S$ specifying, for each role x, which role $g(x)$ one must have access to in order to be able to grant access to the role x. It also reuses the concept of transitivity of access from the level 1 transitive models.

We have now clearly solved the problem of recursive specification of access by making the protection system directly responsible for protecting itself in exactly the same way in which it is responsible for protecting objects. Until now, the access matrix itself has not explicitly been an object. With level 2 systems, we have in some sense moved the mechanism for the protection of the access matrix away from being a '*second layer*' of protection in the commands and into the real world. Suddenly, we are able to use the same code from definition 4.2 on page 46 to protect both the real world objects O and the privileges in the matrix themselves, with o specified by $g(x)$, instead of the more complex code demonstrated in definition 4.5 on page 36.

7.10.7 Summary of The Ideal Model

The level 2 transitive model is our first "*ideal model*" for protection. We may now finally expose the axioms of an ideal protection system.

Axiom 7.1.

1. *Granting a role is a permission on the role.*
2. *Roles control access to multiple objects.*
3. *Roles may possess roles in a transitive relation.*

Axiom 7.1.1 is the canonical solution to the requirements of axiom 2.3. Axiom 7.1.2 introduces the administrative convenience of manipulating large numbers of objects as one role as described in section 7.8. Axiom 7.1.3 allows us to build an arbitrarily deep hierarchy of self management and arbitrarily complex justifications for access, as described in section 7.9.

The truly '*ideal*' nature of this system is only clear when we realise that it satisfies the definition of "*full expressiveness*". Unfortunately, we do not yet have the vocabulary or understanding to state this definition and it languishes in section 12 under the title of "*Further Work*".

The following properties also hold of this system.

- We need no more than 2 rights.
- Practical implementations are fast.
- We have an intuitively correct ordering of roles and privileges.
- Arbitrarily detailed justifications for access and granting may be computed in small time.

We are therefore justified in calling this system our first "*Ideal System*".

7.11 Level 3 Transitive and Higher

If we have completely solved the problem of vetting, then why do higher level systems even exist? At each level, we introduce another apparently unnecessary control right. This question can only be answered by an inspection of implementation: There exist practical systems with operations other than access and grant. A common one is the splitting of a node into a subtree or sub-lattice, as if one were converting the node X in figure 20 on page 122 into a diamond such as that labelled Y.

This operation, involving the creation of principals and the revocation and granting of rights to construct this subtree, itself requires protection. However, as we have shown that the grant operation may be protected using a single right g, so may this new operation, which we call a '*control*' operation, be protected by a single right c.

If the practical system designer wishes to introduce further atomic operations into the protection system, he may protect them each with a single right $d, e, \ldots$. Although the operation described above is safe (provided $g \in [o, x] \geq c \in [o, x]$), this must be shown for each elementary operation proposed.

A case study of a system which introduces further atomic operations into a practical system is given in section 9.13. For now, we conclude this section with the note that two rights is usually enough, but one is not!

7.12 Other Basic Models

There are some models from figure 15 which, while they are off the main path of our exploration, are worth of mention because there are real systems in those categories. The strengths and weaknesses of these systems should be easily identifiable from the classification, and in many cases are readily apparent in the real world. We offer brief notes on some of these systems.

7.12.1 Level 2 Role Based

This category is mentioned only because mechanisms within it occasionally turn up in the wild. It is slightly off the course of our wander through figure 15.

Systems similar to the access control lists used by the Multics "*Discretionary access control*" ([Hon70]) mechanisms may be classified as level 2 role based systems. An access control list entry containing any wildcard elements will specify a role with multiple members (all those user privileges matching the wildcard specification). Each of these entries may be considered a role in that it specifies some logical set of principals permitted to access the file.[53]

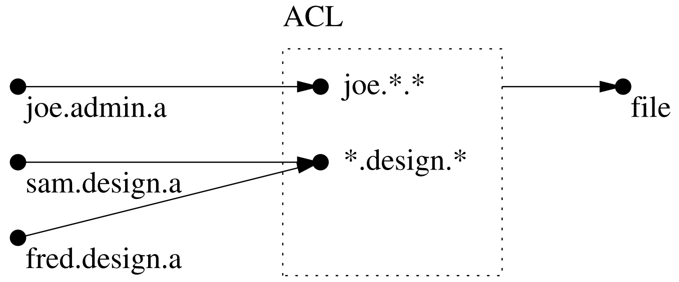


Figure 23: An example Multics ACL demonstrating role semantics

Figure 23 displays an example of a file in a Multics system with an ACL containing two entries. The role `joe.*.*` includes all principals created by the user `joe`, and the role `*.design.*` includes all principals in the `design` project group.

[Hon70] states:

> Users can exercise discretionary control only within those portions of the storage system hierarchy where they themselves have the proper access rights.

The right required to modify the access permissions on a segment is the '`m`' permission of the directory containing the segment. The ACL mechanism makes Multics role-based, and the separate '*m*' function makes it level 2. Therefore Multics may be classified as a level 2 role based system and follows rules similar to those described in section 7.12.3.

7.12.2 Level 0 Transitive

Capability systems including EROS ([SSF99]) and CAP ([NH82], [NW74]) allow transitivity in the access right. If a process can read the capability table of another process, then it can read the 'read' capabilities out of that table, and thus read from any object to which the target process had access. As is likely to be the case with any level 0 model, the model is slightly too simplistic to be entirely practical, and special cases are introduced. For all that, it works very well indeed, as do all transitive models. EROS is described further in section 9.11.

[53] An ACL system which only allows the specification of individual principals would be classified as primitive.

7.12.3 Level 2 Primitive

This might seem a slightly peculiar category, but in fact the rules for the modification of access rights or POSIX ACLs in the Unix filesystem are level 2 primitive in non-sticky directories and (something highly peculiar but similar to) level 0 primitive in sticky directories. Access rights to a file may only be changed if the user has the '`write`' permission to the directory containing the file. The grant role for any file is the directory containing the file: $g(\texttt{/foo/bar}) = \texttt{/foo}$.

This classification is a tricky one given the similarity to the Multics ACLs. The difference between POSIX and Multics ACLs is that entries in POSIX ACLs cannot contain wildcards, and thus cannot specify multiple users within one ACL entry. Some systems classified as role based may easily be mistaken for primitive systems if the entities within the system are labelled differently, and vice versa.

The role based model is computationally distinct from the primitive model only when an object is accessible through more than one role. Otherwise, we may simply consider each role as a composite of many objects, or alternatively, consider the required role as a function of the object.

7.13 Summary

After our failure to build decidably secure protection systems by working from the formalism down, we instead started from the bottom up. We have studied the simplest possible protection systems, all the while maintaining the desirable property of decidability. We have learned that remarkably simple systems are capable of satisfying both our computational requirements of decidability and expressiveness and our business requirements related to ease of administration and delegation of power throughout a true organisational hierarchy.

We have identified an 'Ideal Model', the level 2 transitive model. This model contains only two rights and three commands. From this, we derived the three axioms of an ideal protection system in section 7.10.6. This model has all of the desirable properties described in this section.

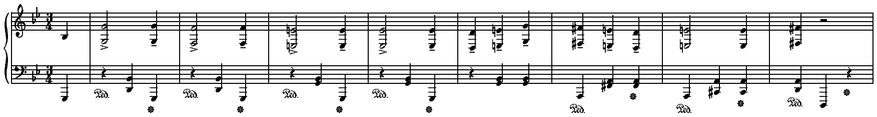
- There need be no distinction between principals, roles and objects.
- Every user need be authenticated for at most one role.
- Two rights is enough, although for practical purposes, we may choose to use more. One is insufficient.
- Principals form a symmetric hierarchy: the diamond pattern.
- Clear and arbitrarily detailed justifications for access may be constructed.
- The answer to an instance of the safety problem is computable in small polynomial time. In fact, a computation of the safety problem is frequently used as the normal access mechanism in these systems, where a good implementation will compute it in at worst linear, and at best constant time!

Material which follows from this section may be found in:

- Section 8: We have not yet studied the current principal, but we have learned many unexpected facts about the nature and relationships between principals and objects in practical systems. Our formal study of the current principal bears out these discoveries to a surprising degree.

- Section 9: The systems which we have studied in this system are elements of a hierarchy of systems. This hierarchy is not artificial, and we rediscover some of these systems in our case studies.
- Section 10: We show that it is possible to build a practical protection system with some extremely desirable properties. This system will be based on the 'Ideal System' found in this section.
- Section 12: We do not yet have the vocabulary to fully express the '*ideal*' nature of the level 2 transitive systems. Much of this vocabulary is introduced in section 8. Some notes on the concept of '*full expressiveness*' appear in section 12, and we note that the ideal systems from this section satisfy that criterion.

Whodunnit . . . ?



8 The Current Principal

8.1 Introduction and Preliminaries

In order to perform a permissions check, we must be able to answer both of the questions posed in section 2.1.3. The first, which thus far has remained unanswered, is, "*Which is the current principal?*". Throughout the discussion, we have assumed some method of identifying the current principal. The ability to answer this question correctly is frequently ignored or underemphasised in matters of security, [HRU76] and following early work discussed in section 6 went so far as to ignore it entirely. It has been researched in significant detail in recent times, as in [PSS01], [FG02], [Ørb95] and [ABHR99].

Definition 2.11 (The Current Principal) states that the current principal is the principal responsible for the execution of the current operation. In the simple world of boxes and keys from example 2.1, the current principal is the principal acting to open a box using a key. It is assumed that no other principal is involved in this operation: this principal is acting only on his own behalf and is therefore entirely responsible for the outcome of his actions. This is not usually the case either in the real world or in computer systems: it is far more common that principals will jointly perform a task.

Without extreme loss of generality within our subject area, we will consider only the cases where one principal asks another to perform a computation and return a value, thus covering the conventional calling semantics for sequential computation as described by the λ-calculus. We will ignore the cases where processes execute in parallel to achieve a task, as described by the π-calculus, although we mention these briefly in section 12.

Within a computer system, operations are carried out by *processes* rather than being performed directly by the user. A *process* is the user's proxy within the computer, an object which is capable of affecting the state of the computer to perform computation. A user will create a process by creating an object containing code and causing that object to be executed. Processes may in turn create processes or perform any other task for which they have the privileges. These code objects, sometimes called '*segments*', are the basic entities responsible for the behaviour of a computation, and may be considered as principals.

Due to the ambiguity surrounding the term *creator* of an object, we will henceforth use two alternative terms:

- The *author* of an object is the principal with control over the behaviour of the object, usually by having write access to the object code or data space.
- The *caller* of an object is the principal responsible for a thread of execution entering an object. This might include the creation of an instance of the object, but does not include any control over the behaviour of the object once invoked.

It would be unfair to place the responsibility entirely on the author of the code for the use that is made of it by the caller.[54] It is also unfair to place the responsibility entirely on the caller for what a piece of code which they call might do.[55] It is clear that *at least both* of these participants are responsible for the consequences.

An object created by any user will represent an extension of the intent of the author and must not be given any privileges higher than those of the author (as required by [Min78] in section 2.4.2). If the privileges of the process are not so restricted, then the author of a code object may break their security sandbox by creating an object which assigns itself a privilege not held by the author and executes code on behalf of the author with these new privileges. (In practice, this frequently means that we must also usually disallow file giveaways to higher privileges, since this gives the appearance that the file was created by a higher privilege object and may participate in a high privilege computation without risk.)

8.2 A Descriptive Introduction to Some Models

We will propose some simple models for identifying the current principal. By understanding the possible attacks against trivial methods, we may recognise vulnerabilities in more complex systems more easily. It will become clear that we must design any system for the identification of the current principal such that information about the involvement of any responsible principal is preserved and presented to the protection system.

Following examples will usually include a malicious principal, a guarded object which performs the security check, and the intermediary used to shield the guard from directly observing the influence of the malicious agent. In each case, the malicious principal would not ordinarily be allowed to perform the protected operation. However, the protection offered by the guard will be bypassed because the underlying system does not provide the guard with the necessary information to know whether the act was originally requested by the malicious principal or the unwitting intermediary.

[54] California does this.
[55] Unix does this.

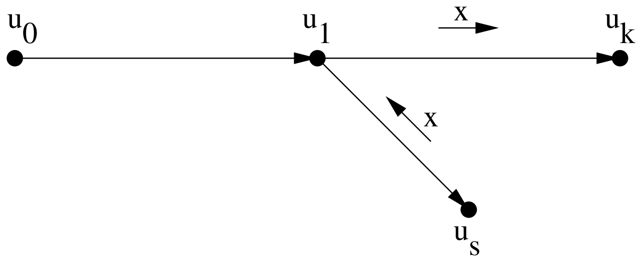


Figure 24: An example call chain with a returned value

We hypothesise a call chain $(u_0, u_1, \ldots u_k)$ in sequential computation. The principal u_k attempts a privileged operation. At that point in the computation, u_k is called the *immediate principal.* u_k has been asked to perform this operation by u_{k-1}, who was in turn called by u_{k-2}. The calling chain goes back to the *root principal* u_0, the principal who originally conceived the call chain. This calling chain $u_0 \rightarrow u_1 \rightarrow \ldots u_k$ will be the basis for the following models.

Figure 24 on page 136 illustrates this example calling sequence which includes the elements used in the following examples. Principal u_0 is the root principal, and asks u_1 to perform an operation. u_1 calls u_s, u_s returns a value x. u_1 then calls u_k, including the value x as a part of the request to u_k. u_k will attempt to access a protected object and is now the immediate principal. The principal u_s is not a part of the call chain at the moment under consideration, having already been removed after returning a value to u_1.

8.3 The Immediate Principal Model

8.3.1 Construction

Let the current principal be the immediate principal u_k. This is valid under the assumption that the immediate principal is able take responsibility for his actions and that he must therefore have enough information to make an informed decision as to whether to allow or disallow an action.

Example 8.1 (The Unchecked Caller (intuitive) ([Har88])). Alice goes to the bank where Bob works. Alice cannot open the safe. Alice asks Bob to give her some money from the safe. Bob uses his key to open the safe and gives Alice the money.

Formally, with $k = 1$: Alice is the root principal, u_0. Bob is the immediate principal u_1. The protected operation was performed using the privileges of u_1.

8.3.2 Explanation

The safe is a protected object. The protection system checks only that the immediate principal (Bob) has the privilege (the key) to access the safe, since the immediate principal is assumed to be the responsible principal.

In the real world, Bob would first check before opening the safe for Alice whether Alice is a senior employee of the bank or otherwise has the rights to open the safe. However, Bob would still have no way to verify that Alice was responsible in turn for her own request to Bob unless he trusted Alice entirely to perform this verification. If Alice *was* asked by another principal, then she would in turn have the same problem, and everybody must trust everybody else to verify *correctly* the responsible root principal.

By asking the immediate principal to make a security decision about its callers, we are requiring that the immediate principal (and possibly every principal in the call chain) reimplement the system for identification of malicious principals. Thus we gain nothing, save passing off the problem to someone else. Also, somewhere in the call chain, someone is probably lying.[56] It is still more likely that someone will perform their job incorrectly and the system will break, even when implemented with the best intentions.

8.3.3 Examples

Unix uses this model for inter-process communication and set-user-id (SUID) binaries ([PASC97] §3.1.2.2: "*exec*", [OG03]: "*exec*"). Exploits for this model involve entering high privilege code from low privilege code over a network, over a socket, by RPC or by executing a SUID binary with unchecked input.

Earlier versions of the XFree86 window system server executed with root privilege and could be invoked with a user-specified configuration file. If the configuration file contained an invalid line, the line would be printed and the process would terminate. It is not possible for a Unix user to read the file `/etc/shadow` since it contains hashed passwords. However, invoking the X server with `/etc/shadow` as a configuration file would cause the X server to print the root password hash for the system, which could then be cracked in the usual manner.

The NFS daemon ([NWG03]) must also check the address of the calling client machine before allowing an NFS export to take place. An unchecked NFS mount effectively grants the client machine a ticket allowing it to request modifications to files in the store of the called server machine.

It is the responsibility of the target process in this model to verify the intent of the request in any way available to it. Inadvertent execution of, or buffer overruns in, privileged processes are examples of failures in this model.

8.4 The Root Principal Model

8.4.1 Construction

Let the current principal be the root principal u_0. This is valid under the assumption that the root principal is able to take responsibility for all his callees. It seems more feasible that a principal will have knowledge of its callees than of its callers since the identities of callees are frequently embedded in code. However, many of the same problems arise.

Example 8.2 (The Unchecked Callee (Trojan Horse) (intuitive)). Alice happens to walk into the bank where Bob works. Alice cannot modify the account books. Bob says to

[56] Other than all the lawyers.

Alice, "*Please keep our accounts for us*" and gives her the account books. Alice transfers all the money to her account, leaves the bank, withdraws the cash and goes to Cuba.

8.4.2 Explanation

Formally, with $k = 1$, Bob is the root principal, u_0. Alice is the immediate principal u_1. The protected operation was performed using the privileges of u_0.

Now the account book is a protected object. The protection system checks only that the root principal (Bob) has the privilege (possession thereof) to modify the account book, since the root principal is assumed to be the responsible principal.

In the real world, Bob should check first that Alice is a responsible accountant before asking her to maintain the books. However, in this case the reverse of the previous example applies and Bob has no way to know whether Alice will in turn delegate the task to an irresponsible accountant. Again, a lot of trust must exist and no mistakes may be made by anybody in the chain.

8.4.3 Examples

In some recent versions of Microsoft Windows, when an application fails, an application called *DrWatson* is executed with `SYSTEM` privileges to record information about the failure. The name of the application to execute on failure is stored in the registry, and may be modified by any user. By modifying the value in the registry and then crashing any program[57], an arbitrary user may execute any program with `SYSTEM` privileges. It is the responsibility of any process in this model to verify the intent of any program it executes as a child process. The ability to cause a high privilege process to execute a file of choice will lead to a breach of security.

8.5 Aside: Combining Principals

There are clear weaknesses in identifying as the current principal either the immediate or the root principal, or, for that matter, any other single principal.

While from an external and intelligent viewpoint, we may make a judgement as to which of many participants was the primary instigator of a malicious act, this is an intuitive judgement and not necessarily correct. Any participant may tell lies to, or about the actions of, any other participant, and thus there exist cases where any participant may be responsible for an outcome.

We must design a concept of joint responsibility. A principal is merely a set of permissions, by definition 2.9. By deciding what permissions the partnership must have, we can define a new principal to represent the partnership and identify that as the current principal.

Let Alice and Bob cooperate on a task in such a way that each may influence the work of the other. Consider a permission a held by Alice but not by Bob, and a permission b held by Bob but not by Alice. If the partnership has permission a then Bob has access to permission

[57] This isn't hard either.

a, albeit possibly indirectly by exerting control over the partnership. This is not desired since then Bob would be in a position to exercise permissions to which he is not entitled, therefore the partnership must not hold permission a. Equivalently, the partnership must not hold permission b, nor may it hold any permission not held by either Alice or Bob. Those permissions which are left are those held by both Alice and Bob, the intersection of their permission sets.

Interestingly, this gives us an immediate corollary for the safety problem for cooperative partnerships: If a joint operation between principals is constructed using the intersection of privileges between all principals involved, then no right is unsafe which was not unsafe for *each and every* involved principal.

8.6 The Stack Model

8.6.1 Construction

As we have seen, any principal in the call chain could conceivably pervert the outcome of a call and cause some undesired operation. Therefore, we construct a system where each and every principal in the call chain is jointly responsible for the action performed. In this case, each and every principal must have the privilege to perform any operation executed by the chain.

The current principal in the stack security model is an artificial, constructed principal the permissions of which are the intersection of the permissions of every principal in the call chain.

$$\text{privileges}(\text{current\_principal}) = \bigcap_{i=0}^{k} \text{privileges}(u_i)$$

Stack security suffers from a more subtle variant of the Trojan horse weakness. While the stack security model is safe against malicious code in the call chain, it is still possible for a piece of malicious code to return untrusted data into a call chain. A piece of data is not usually associated with a principal: it is not executable and is in itself inactive. However, it may be designed with intent and must be considered as a part of the computation.

Example 8.3 (The Unchecked Value (intuitive)). Carol is the guard at the door of the House of Commons. She must invite members of parliament to enter but prevent anyone else from entering. A tourist approaches. Carol asks the tourist, "*What is your name?*" The tourist answers, "*Tony Blair!*[58]" The guard looks on her list of Members of Parliament, finds the name and opens the door for the tourist.

Formally, with $k = 0$, Carol is both the root and the immediate principal, u_0: the call chain contains only Carol at the point when the protected operation is performed. The tourist is a principal u_s, also illustrated in figure 24. u_0 calls u_s. u_s returns a value and is then no longer on the stack. The protected operation was performed using the privileges of u_0.

[58] The British prime minister at the time of writing.

8.6.2 Explanation

The House of Commons is protected. The protection system on the door checks that the guard (Carol), being the only principal on the call chain, may open the door. However, Carol made a decision based on false information provided by the tourist.

In the real world, Carol should first verify that the tourist really *is* Tony Blair. However, this may be impossible unless some third party can vouch for any data provided by the tourist without falling into the same trap.

8.6.3 Examples

The Majordomo mailing list manager, which executes with raised privilege, takes as an input argument the name of a configuration file to read in. Any user may execute Majordomo with a configuration file of their choice, and with a suitably crafted configuration file, may gain arbitrary access to the raised privilege used by Majordomo. It is the responsibility of any process in this model to verify the suitability of any data it uses in computation.

[FG02] presents a very good discussion of stack inspection security models. Stack inspection is used by Java, the .NET CLR, some LPC models and many others.

8.7 The Data Model

8.7.1 Construction

Data inspection security combats the main remaining vulnerability of the stack security model, the unchecked return value, as exposed in example 8.3. We must be able to trace the principals responsible for all data participating in the computation as well as code on the direct call chain.

The information about the principals responsible for any piece of data must be attached to the piece of data throughout its lifetime. This information will contribute to the calculation of a current principal in any context. The data model is harder to describe than the stack model since it is not possible to identify the source of a piece of data from a snapshot of a computation unless this auxiliary information has been maintained throughout the computation.

By the logic in section 8.5, if we combine two pieces of data in a computation, the principal considered responsible for the result will be an artificial principal whose privileges are the intersection of the privileges of those of the principals responsible for the input data. The current principal in any request for a protected operation will not exceed the intersection of the privileges of all the arguments passed to the request for the protected operation. There are several variants of data inspection calculus which include or exclude the responsibility of various possible participants.

This will be the model with which we introduce our formal calculus; we defer further description of the possible models until section 8.12, where the precise relationships between models will be explained in more detail.

8.7.2 Examples

Simple data tainting models, as that in Perl or JavaScript, have only one privilege: '*untainted*', and two principals: '*untainted*' and '*tainted*'. The *untainted* principal holds the *untainted* privilege; the *tainted* principal holds no privileges, thus there is no ambiguity of terminology. Any operation involving tainted data produces a tainted result, and no protected operation may be carried out using tainted data. Every piece of data has an associated *taint bit*, which is set if the data is tainted, in other words, if the data does not hold the *untainted* privilege.[59] In Perl, protected operations which may not be performed with tainted data include *execute process* and *write file*. In JavaScript, data tainting is implemented as a privacy measure rather than a security measure; tainted data may not be transmitted across the network ([NCC99]):

> When data tainting is enabled, JavaScript in one window can see properties of another window, no matter what server the other window's document was loaded from. However, the author of the other window taints (marks) property values or other data that should be secure or private, and JavaScript cannot pass these tainted values on to any server without the user's permission.

8.8 Aside: The Dangers of Ad-Hoccery

It is possible to cope to some extent with this and other like scenarios by performing exhaustive identification at every step of the source of all security-critical data which might influence the outcome of a computation, but data processed by a program is frequently mistakenly not considered to be security-critical. An accurate identification of which data is relevant to security and which is not can be difficult, especially in the absence of knowledge about which privileges will be requested or what will be done with data merely passed on by a piece of code.

Security issues rapidly become an every day part of the life of the ordinary programmer and any verification becomes ad hoc where it is even possible. This is not a desirable circumstance; the average programmer is little aware of security and prone to errors and inadvertent mistakes. We wish to identify automatically that information which is relevant to security and perform security checks without assistance from or the intervention of the application programmer.

In this section, we will study which principals may be responsible for the outcome of a computation and what information we must automatically preserve in order to identify a current principal at any point in a computation. This side computation could be performed adequately and possibly even correctly by the user application. However, in practice it will usually be performed incorrectly, even when implemented with the best intentions. We aim therefore to perform all security related computation in the trusted computing base, thus keeping all security checking code under the highest scrutiny and avoiding duplication of effort throughout applications.

[59] This slight substitution of terminology is made for reasons of convention and consistency with existing implementations. The mistake was on the part of the implementors, who originally assumed a "secure by default" policy and only set the taint bit for known tainted data.

8.9 Indemnification

We may build a perfect world where every principal involved in any way in a computation is responsible for the outcome of that computation. Each responsible principal is accounted for when privileges are checked. While this model is ultimately unbreakable, it becomes immediately unworkable in practice.

Example 8.4 (Without Indemnification). Alice goes to the bank where Bob works.

- She tries to remove £50 from the bank. She cannot, for the money is in the safe and only Bob can open the safe.

- She says to Bob, "*Please give me £50.*". Bob looks in his ledger and finds that Alice's account contains £100. Bob cannot update the ledger or open the safe because Alice was partly responsible for his actions and Alice does not have the privilege to update the ledger or open the safe.

Alice may not withdraw money from the safe by asking Bob, nor may Bob ask Alice whether she wants money in order to give it to her because in both cases Alice would have participated in the computation, and since she does not have the privileges to withdraw money, the privilege check on the safe would deny access.

However, Bob is capable of verifying that the request is not malicious as far as he and the safe are concerned: Alice is withdrawing money within her budget and from her own account. The desired sequence of events might be as follows:

Example 8.5 (With Indemnification). Alice goes to the bank where Bob works.

- She says to Bob, "*Please give me £150.*". Bob looks in his ledger and finds that Alice's account contains only £100. Bob says, "*No, you only have £100.*"

- She says to Bob, "*Please give me £50.*". Bob looks in his ledger and finds that Alice's account contains £100. Bob decides that this request is valid and proceeds using only his own privileges. He updates the ledger, opens the safe and gives Alice £50.

Alice is still responsible for the action which is performed, that of removing £50 from the safe, but she is not permitted to do so directly. Even when she asks Bob to act as an intermediary, an accurate computation of responsibility by the safe door would include Alice as a caller and hence a major participant and would deny access to Bob because he was asked by Alice.

In our ideal world, Bob inspects Alice's request according to some criteria of his own before *using his own privileges* to remove Alice's label of responsibility from the transaction and allow the operation to continue. Bob has thus indemnified the transaction against interference from Alice. As we formalise our methods for identification of the current principal, we will include a primitive for performing an indemnification.

A system which equates the current principal with the immediate principal is equivalent to a system which performs this indemnification on every call. We have shown such a model to be weak by example, but now we have also shown that there are times when it is a requirement for a practical system that we be able to perform an indemnification. These indemnifications must be carefully designed, since they will be the obligatory weak point in a system which may otherwise be perfectly designed and executed.

8.10 Partial Principals: A New Foundation for Computation

Our current objective is to identify a *principal* s about whom we may ask, "*Does s have the permission to access a given object o?*" We have gained much insight into the representation of a principal as a set of permissions, but is it even necessary to identify a current principal? How much work can we save? Usually, quite a lot.

At no point have we ever *needed to know* who is the current principal other than in order to discover whether the current principal has a given privilege. If it is possible to identify that the current principal must have that privilege without totally evaluating the principal, then we may save time and effort in the protection system. We will therefore try to identify classes of principal such that certain permissions are present within a class. We will achieve this using *partial principals*: principals about which only partial information may be available.

Let the set of all permissions again be denoted $\mathbb{P}$. A full principal P is a set of permissions $P \subseteq \mathbb{P}$. Equivalently, we may view P as a map from $P : \mathbb{P} \to \{\text{true}, \text{false}\}$.

A partial principal D represents partial information about a principal $P \subseteq \mathbb{P}$. For each $p \in \mathbb{P}$, we may know that $p \in P$, we may know that $p \notin P$, or we may have no knowledge of whether $p \in P$.

Let $\mathbb{K} = \{\text{true}, \mho, \text{false}\}$ represent the three states of knowledge where *true* represents knowledge of truth, *false* represents knowledge of falsehood, and $\mho$ represents lack of knowledge. $\mathbb{K}$ is a linear order false $\leq \mho \leq$ true. We make the natural definitions of $\wedge, \vee, \Rightarrow, =: \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ in this three-valued logic:

$\wedge$	true	$\mho$	false
true	true	$\mho$	false
$\mho$	$\mho$	$\mho$	false
false	false	false	false

$\vee$	true	$\mho$	false
true	true	true	true
$\mho$	true	$\mho$	$\mho$
false	true	$\mho$	false

$=$	true	$\mho$	false
true	true	$\mho$	false
$\mho$	$\mho$	$\mho$	$\mho$
false	false	$\mho$	true

$\Rightarrow$	true	$\mho$	false
true	true	$\mho$	false
$\mho$	true	$\mho$	$\mho$
false	true	true	true

We note that $a \Rightarrow b \equiv \neg a \vee b$ and $a = b \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$ as usual. Interestingly, $\neg\mho = \mho$ in this logic.

Lemma 8.1. *The structure* $(\mathbb{K}, \wedge, \vee)$ *is a lattice*[60] *with meet* $\wedge$ *and join* $\vee$.

Proof. The proof is immediate from the truth tables above.

□

Definition 8.1 (Partial Principal). Given a principal $P \subseteq \mathbb{P}$ about which we have partial information, the partial principal D_P is a map $D_P : \mathbb{P} \to \mathbb{K}$ where the map is defined elementwise on $\mathbb{P}$:

$$D_P(p) = \begin{cases} \text{true} & \text{if} \quad p \in P \\ \mho & \text{if} \quad \text{it is not known whether } p \in P \\ \text{false} & \text{if} \quad p \notin P \end{cases}$$

Henceforth we will ignore the hypothetical full principal P about which D_P represents partial information and write D without a subscript.

Definition 8.2 (Operations on the Partial Principal).

We define equality of D and E as a conjunction of equalities of elements in $\mathbb{K}$, giving a result in $\mathbb{K}$:

$$(D = E) := \bigwedge_{p \in \mathbb{P}} D(p) = E(p)$$

We define union and intersection elementwise:

$$\begin{aligned} (D \cap E)(p) &= D(p) \wedge E(p) \\ (D \cup E)(p) &= D(p) \vee E(p) \end{aligned}$$

The definitions of union and intersection match our intuitive concepts of union and intersection of partial information. The definition of equality is defined elementwise in $\mathbb{K}$ using operators in $\mathbb{K}$.

Lemma 8.2. *The set of partial principals form a lattice* $(\{D : \mathbb{P} \to \mathbb{K}\}, \cap, \cup)$.

Proof. The proof is immediate from the definitions and truth tables above.

□

Definition 8.3. We define a slightly unusual ordering operator on this lattice:

$$D \subseteq E \overset{\text{def}}{\Longleftrightarrow} D = D \cap E$$

or equivalently:

$$D \subseteq E \overset{\text{def}}{\Longleftrightarrow} D \cup E = E$$

Since the range of our equality operator is $\mathbb{K}$, the range of the subset operator is also $\mathbb{K}$. If $D \subseteq E = \text{true}$, we say that E "*satisfies*" D.

[60] See appendix A.6.

Notation. We say $p \in D$ if $D(p) = \text{true}$ and $p \notin D$ if $D(p) = \text{false}$.

We will define $D(S) = \{D(p) | p \in S\}$ for any full principal S. A partial principal D is equivalent to some full principal if $\mho \notin D[\mathbb{P}]$; conversely a full principal $S \subseteq P$ is a partial principal such that $S(p) = \text{true}$ for all $p \in S$ and $S(p) = \text{false}$ otherwise. It therefore makes sense to write $D \cup S$, $D \cap S$ *et cetera* where D is partial and S is full or vice versa. We will consider this to be implicit when we perpetrate certain abuses of notation. It will not always be necessary to state whether a principal is partial or full.

We must now verify that this subset operator satisfies our intuitive requirements for the operator. We may evaluate the current principal as a partial principal D. Given a (full) principal R which we are required to satisfy, we want $R \subseteq D$ to mean that every privilege which is in R is individually satisfied by D, in other words $p \in R \Rightarrow D(p)$. The subset operator will take a value in $\mathbb{K}$ where $\mho$ means that we do not yet have enough information to make a decision. In fact our operator satisfies a stronger property in that we may allow R to be a partial principal and still obtain behaviour which matches our intuition.

$$\begin{aligned} R \subseteq D &\implies R = R \cap D \\ &\implies R(p) = R(p) \wedge D(p) \qquad (\forall p \in R) \\ &\implies R(p) \Rightarrow D(p) \qquad \text{(by the truth tables above)} \end{aligned}$$

Notation. We will use $\emptyset$ as the special partial principal such that for each $p \in \mathbb{P}$, $\emptyset(p) = \text{false}$. This partial principal is equivalent to the full principal $\emptyset$, so ambiguity is unimportant.

The partial principal is a powerful tool in what follows, since it frequently allows us to derive *just enough* information to make a security decision without having knowledge of the full context of a particular operation in order to evaluate a full current principal. This allows us to perform reductions of subexpressions at compile time and to analyse the effects of optimisations on the reliability of the system.

8.11 The λ_{sec}-calculus

The λ-calculus ([Bar84], [HS86] and others) is a foundation for logic in sequential computation. It may be used to model the user computation within which we wish to enforce protection. Familiarity with the λ-calculus will be assumed.

Since any algorithm for the identification of the current principal will be performed in the trusted computing base (alongside any computation performed by the user-visible mutator), these algorithms will be expressed using extensions to the λ-calculus with semantics describing transformations performed in parallel with the usual λ-calculus reductions.

We choose to formalise our algorithms for manipulating principals using a variant of the untyped λ_{sec}-calculus ([FG02], from [PSS01]), a call by value λ-calculus. We introduce a new extension to this traditional λ_{sec}-calculus, the notion of the partial principal. We will perform our computations using partial principals where previously full principals have been required.

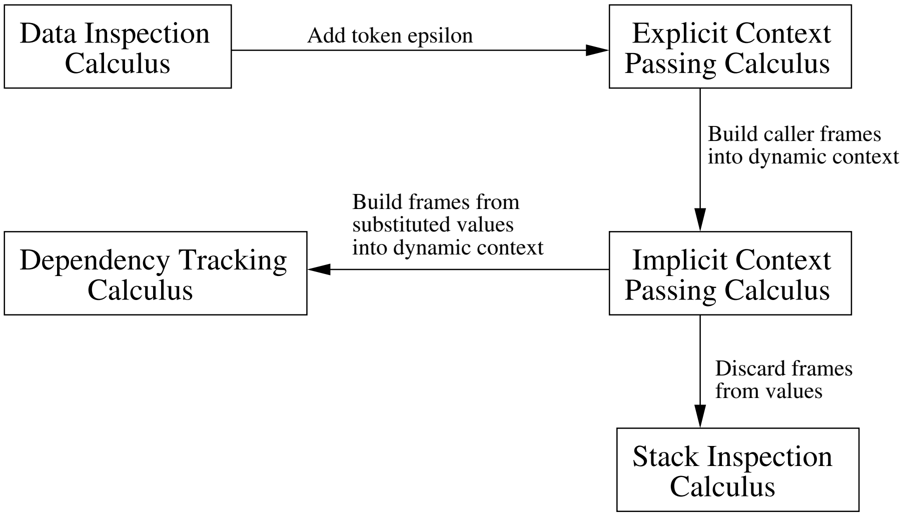

Data Inspection Calculus
Add token epsilon
Explicit Context Passing Calculus
Build caller frames into dynamic context
Build frames from substituted values into dynamic context
Dependency Tracking Calculus
Implicit Context Passing Calculus
Discard frames from values
Stack Inspection Calculus

Figure 25: The relationships between our λ_{sec}-calculi

We then build compatible calculi for describing common algorithms such that we may compare and combine the operational semantics of the calculi to derive variants of the algorithms. To this end, we establish a basic set of definitions and semantics, inoperable in itself, which we will augment with further rules to describe a particular model. The approximate roadmap for our calculi is displayed in figure 25; it includes brief notes on the relationships between the calculi.

8.11.1 Basic Expressions in λ_{sec}-calculus

Basic expression syntax is displayed in figure 26. The basic λ-calculus expressions need little or no explanation beyond that given in [Bar84].

The first major addition to the λ-calculus is the concept of a '*framed*' expression. The '*frame*' R in $R[.]$ is the principal immediately responsible for the enclosed expression. Every expression is owned by some principal, usually the principal who created the object or file containing that code; this principal is immediately responsible for the actions of that expression. This responsibility is denoted by the appearance of the principal as a frame on the expression. We write $\mathbb{P}[e]$ for an expression in the trusted computing base or TCB ([Shi00]).

In any context, the principal of the innermost frame of the context is the "*immediate principal*", the principal immediately responsible for the current expression.

We subscript the λ operator with a principal P. Informally, this is intended to perform the expected β-reduction if and only if the full principal P is satisfied by the current principal or thread of execution, and fail otherwise. This may be used to guard entry points into privileged code. An expression of the form $\lambda v.\mathbb{P}[e]$ represents code executing in the TCB with full access to all objects. Writing $\lambda_P v.\mathbb{P}[e]$ expresses a restriction to the principal P for the substitution of v into e. It will usually be desired that the principal P is satisfied by all parties who could affect the execution of e. A subscripted λ is said to lead to a "*guarded substitution*".

Permissions and principals

$p_0, p_1 \in \mathbb{P}$	permissions
$P, R, S, T \subseteq \mathbb{P}$	principal: a set of permissions
$D, E : \mathbb{P} \to \mathbb{K}$	partial principal

Expressions

e, f ::=	expression
x, y	variable
$\lambda_P x.e$	function
ef	application
fail	failure
$R[e]$	framed expression
grant R in e	indemnification of the caller
untaint R in e	indemnification of the callee

Values and framed values

$u, v ::= x \mid \lambda x.e \mid \mathsf{fail}$	value
$w ::= v \mid R[w]$	framed value

Abbreviations

$\lambda x.e \mathrel{\widehat{=}} \lambda_{\emptyset} x.e$
$\lambda_P xy.e \mathrel{\widehat{=}} \lambda_P x.\lambda_P y.e$
$\lambda_P\_.e \mathrel{\widehat{=}} \lambda_P x.e$ for any $x \notin \mathrm{fv}(e)$
$v \mathrel{\widehat{=}} \mathbb{P}[v]$
$PS[] \mathrel{\widehat{=}} P[S[]]$
ok $\mathrel{\widehat{=}} \lambda x.x$
$e; f \mathrel{\widehat{=}} (\lambda\_.f)e$

Figure 26: Basic expression syntax

We introduce two other extensions to the syntax for expressions to cover two types of indemnification (as introduced in example 8.5): indemnification of the caller by *grant*: "*Alice told me to do it but I've checked her out and she's OK*"; and indemnification of a value returned by a callee by *untaint*: "*Bob told me this but I've checked it and it's OK*". We therefore add the *grant* operation for indemnification of the caller to the callee, and the *untaint* operation for indemnification of values to the caller. Since the indemnifying principal may only indemnify information up to its own privilege level, both of these operations may add only privileges held by the current immediate principal.

As in [Bar84], we use the notation $\mathrm{fv}(e)$ to denote the set of free variables in an expression e.

8.11.2 Partial Context

Our definition of context is similar to that in [Bar84], with the addition that frames around expressions form a part of context. Security context in our calculus is bipartite and denoted $\langle {S \atop D} \rangle[]$ where S is the immediate principal. D denotes the dynamic context, which we develop into a designation of responsibility within the context. Reduction within a context $\langle {S \atop D} \rangle[]$ is denoted $\to^S_D$, as in [FG02]. However, unlike this previous presentation, D will usually be partial. [FG02] also makes reduction conditional on $D \subseteq S$, but in our calculus, reductions within a context are unconditional, by theorem 8.3 to come. The semantics of contexts will describe how framed expressions $R[e]$ interact with contexts $\langle{S \atop D}\rangle[e]$.

Contexts

$C[] ::=$	contexts
$\langle{S \atop D}\rangle[]$	context
$e \rightarrow^S_D e'$	contextual reduction

Figure 27: Context syntax

8.11.3 Basic Operational Semantics

The basic reduction rules from figure 28 will be implicitly included in each ruleset we define.

Operational semantics (basic)

Basic Ctx Rator	$\dfrac{e_1 \rightarrow^S_D e_1'}{e_1 e_2 \rightarrow^S_D e_1' e_2}$
Basic Fail Rator	$\mathsf{fail}\, e \rightarrow^S_D \mathsf{fail}$

Figure 28: Basic operational semantics

The expression *fail* is a discriminated failure generated by a security exception. We do not require that evaluation terminate immediately on the generation of a *fail* by introducing a rule

$$\text{Basic Fail Rand} \qquad e\, \mathsf{fail} \rightarrow^S_D \mathsf{fail}$$

as in [FG02]. If the *fail* appears as the operand of $\lambda\_.e$, it will not participate in the outcome and the reduction may yet succeed. This addition was very likely modelled on the Java implementation in which the generation of a discriminated failure is represented by throwing an exception, which would indeed cause the thread to terminate.

8.11.4 Overall Context of Execution

The system, in causing the evaluation of an expression, is itself a root principal for the call chain. Since all principals on the call chain form a part of the context of execution and hence affect security decisions made within the expression, we must consider which principal to use for this root principal.

There are two obvious choices for the root principal: $\emptyset$ and $\mathbb{P}$. Given an expression e, we may evaluate $\langle{\emptyset \atop \emptyset}\rangle[e]$ to deny permission by default or $\langle{\mathbb{P} \atop \mathbb{P}}\rangle[e]$ to allow permission by default. Netscape 4.0 chooses the former whereas Sun JDK 1.2 and Microsoft's IE 4.0 choose the latter ([Wal99]). e will also be framed by the object responsible for it, so even in the case that $\mathbb{P}$ is the root principal, the contextual privileges will be reduced almost immediately. Since we are constructing contextual equivalences, it is only important that we require $D \subseteq S$ in any top level context $\left\langle {S \atop D} \right\rangle[]$. This will form the base case for a later proof by induction that $D \subseteq S$ in every context $\left\langle {S \atop D} \right\rangle[]$.

8.11.5 Responsibility in λ-calculus

Throughout the introduction to this section, and in all the examples following section 8.2, we have seen malicious principals influencing a computation to their own ends. We consider these principals to be in some sense '*responsible*' for the outcome. If we are to compute a single current principal '*responsible*' for an outcome, as in definition 2.11, we must have a formal concept of responsibility which can be identified and manipulated in a λ_{sec}-calculus. Therefore we make the following key conceptual definition.

Definition 8.4 (Responsibility in λ-calculus). Given a lambda-expression and a reduction to an outcome, every principal framing a sub-expression which participates in the outcome is responsible for that outcome.

8.12 Data Inspection Calculus

The data inspection calculus is the fundamental calculus of responsibility. It tracks all responsible principals in all expressions, and makes a good base upon which to build further calculi.

8.12.1 Construction

Given an example redex $\lambda x.\mathbb{P}[e]$ in the trusted computing base, we wish to control who may influence the evaluation of any protected subexpression of an expression e. The only way to influence the evaluation of e is to perform a β-reduction in e by substituting a value for x.

We define semantics for $\lambda_P x.e$ with a framed operand to ensure that only values generated by principals satisfying P may be substituted by this guarded substitution. A value w supplied by any untrusted principal, either directly or indirectly, must not be substituted into e by the λ_P.

For example, to protect a filesystem, we may require that the principal F (for filesystem) is satisfied before allowing a file to be read. We would therefore construct an expression $\lambda_F x.\mathbb{P}[\text{read}(x)]$; only a value generated only by principals satisfying F may affect the evaluation of this expression. A request involving an untrusted principal who did not satisfy F should not be able to generate a filename w satisfying F.

We must therefore be able to identify all principals who may have either directly or indirectly affected a given operand w. Our semantics for the manipulation of frames cause any value to retain the frames of all principals having affected the computation of the value.

In order to control effectively who may substitute framed values into e, $\lambda x.\mathbb{P}[e]$ must also be a *closed* lambda term. Were this not so, it would be possible to substitute an arbitrary value using an expression of the form $\lambda y.\lambda_P x.\mathbb{P}[e]$ where $y \in \text{fv}(e)$.

8.12.2 Formalism

The only relevant element of context is the most recent frame, the immediate principal. Therefore, we use $\to^{S}_{.}$ to represent a reduction in the context of the immediate principal, where the . is a placeholder for a dynamic context to be introduced in a later model.

Data Ctx Rand	$\dfrac{e_2 \quad \to^{S}_{.} \quad e_2'}{w_1 e_2 \quad \to^{S}_{.} \quad w_1 e_2'}$
Data Ctx Frame	$\dfrac{e \quad \to^{R}_{.} \quad e'}{R[e] \quad \to^{S}_{.} \quad R[e']}$
Data Red Grant	$\text{grant } R \text{ in } e \to^{S}_{.} e$
Data Ctx Untaint	$\dfrac{e \quad \to^{S}_{.} \quad e'}{\text{untaint } R \text{ in } e \quad \to^{S}_{.} \quad \text{untaint } R \text{ in } e'}$
Data Red Untaint Frame	$\text{untaint } R \text{ in } P[w] \to^{S}_{.} (P \cup (R \cap S))[\text{untaint } R \text{ in } w]$
Data Red Untaint Value	$\text{untaint } R \text{ in } v \to^{S}_{.} v$
Data Red Frame Rator	$R[w_1]w_2 \to^{S}_{.} R[w_1 S[w_2]]$
Data Red Frame Rand	$(\lambda_R x.e)P[w] \to^{S}_{.} \begin{cases} (\lambda_R x.e[x := P[x]])w & \text{if} \quad R \subseteq P = \text{true} \\ \text{fail} & \text{if} \quad R \subseteq P = \text{false} \\ \text{(not reducible)} & \text{if} \quad R \subseteq P = \mho \end{cases}$
Data Red Appl	$(\lambda_R x.e)v \to^{S}_{.} \begin{cases} e[x := S[v]] & \text{if} \quad R \subseteq S = \text{true} \\ \text{fail} & \text{if} \quad R \subseteq S = \text{false} \\ \text{(not reducible)} & \text{if} \quad R \subseteq S = \mho \end{cases}$

Figure 29: Data inspection operational semantics

The reduction rules for this calculus are displayed in figure 29, and as with all our calculi, the basic rules from figure 28 are implicitly included. We highlight some of these new rules for discussion.

- **Data Ctx Rand** is an extension to the traditional lambda calculus allowing the reduction of the operand of a framed value.
- **Data Ctx Frame** shows that a change of frame changes the context of execution by changing the immediate principal.
- **Data Red Grant** is a dummy reduction since we take no interest in the caller, merely in what we have been asked to do. We are therefore never required to indemnify the caller.
- **Data Red Untaint Frame** is an indemnification by the immediate principal S of a value w, which has presumably been verified 'correct' in some sense by the immediate principal. This rule is the step of an induction over the number of frames around the untainted value; on each step one frame is "filtered" through the untaint and the result is an untainted value.
- **Data Red Untaint Value** is a reduction since the untaint is not relevant on an unframed (and hence trusted) value. This rule is the base case of the induction over frames.

- **Data Red Frame Rator** may look a little unusual. The primary purpose of this rule is to tag the return value of a redex with the frame R.

  Perhaps the most obvious alternative to this rule would be

  $$R[w_1]w_2 \to^{S}_{\cdot} R[w_1 w_2]$$

  This has an immediate failure in that the value w_2, which could have been provided as a constant by S, has now left the immediate context of S with no tag designating S as responsible for it. Therefore, on removing w_2 from the immediate context $S[]$, we must add a frame around w_2 to identify the source of the value and thus avoid trivial Trojan horse values.

  We cannot simply do

  $$R[w_1]w_2 \to^{S}_{\cdot} w_1 R[S[w_2]]$$

  since this would not make the frame R available to the callee in the case that the reduction of the expression w_1 does not depend on the operand. In this case the frame R, representing a principal which certainly has an influence on the computation by choosing w_1, would not be available to any guarded substitution wishing to verify the source of the eventual reduct.

  Another choice would be

  $$R[w_1]w_2 \to^{S}_{\cdot} R[w_1 R[S[w_2]]]$$

  to make the frame R available for any guarded substitutions within w_1, but rule *Data Red Appl* and future applications of *Data Red Frame Rator* cover this case and make the frame R available for any guarded substitutions involving w_2 inside w_1, so the added complexity is unnecessary.

- **Data Red Frame Rand** checks frames on the operand before performing a guarded substitution. It is to be noted that without regard to whether x is free in e. This feature may be used to check a dummy argument for controlling access to a TCB entry point which takes no arguments. This rule and the one following form respectively the step and the base of another induction over the number of frames on the operand of the application.

- **Data Red Appl** makes the immediate principal responsible for any unframed (otherwise implicitly trusted) expression which may be substituted into a guarded expression.

  If a reduction is not possible in $\langle^{S}_{\cdot}\rangle[]$ then we must appeal to a larger context to gain more information about D.

Rules *Data Red Frame Rator* and *Data Red Appl* between them ensure that any term retains a frame from the immediate principal responsible for it when it is exported from the original frame. In this way, correctness of the frame on any expression is guaranteed since any expression will at all times have a frame around it representing each principal by which it has been framed. No principal may manipulate an expression which it does not frame.

8.12.3 Consequences

The data inspection system allows the evaluation of a subexpression requiring privileges even if called from a non-trusted environment as long as the evaluation of that subexpression cannot be influenced by the caller. We will term these trusted subexpressions '*side expressions*' since they are not directly influenced by the arguments given.

For example, a subsystem allowing access to a privileged subsystem may wish to record access into a log file requiring privilege $\mathbb{P}$ to write via a guarded function *log*. Let

$$\text{log} \mathrel{\widehat{=}} \lambda_{\mathbb{P}}\_.\mathbb{P}[\text{do\_log}]$$

Assume in the following that $P \subseteq R$ and that P is not a superuser (so has privileges less than $\mathbb{P}$).

$$\begin{aligned}
(\lambda_R x.\mathbb{P}[\text{log ok}; \text{foo } x])P[v] &\overset{\text{Data Red Frame Rand}}{\Longrightarrow} (\lambda_R x.\mathbb{P}[\text{log ok}; \text{foo } P[x]])v \\
&\overset{\text{Data Red Appl}}{\Longrightarrow} \mathbb{P}[\text{log ok}; \text{foo } P[v]] \\
&\overset{\text{Definition of log}}{\Longrightarrow} \mathbb{P}[(\lambda_{\mathbb{P}}\_.\mathbb{P}[\text{do\_log}])\,\text{ok}; \text{foo } P[v]] \\
&\overset{\text{Data Red Appl}}{\Longrightarrow} \mathbb{P}[\mathbb{P}[\text{do\_log}]; \text{foo } P[v]]
\end{aligned}$$

which then continues to reduce normally. However, the function *log* may not be called in the immediate context of P because P is not capable of generating any argument which may be substituted into *log* by $\lambda_{\mathbb{P}}$.

$$\begin{aligned}
log P[v] &\overset{\text{Definition of log}}{\Longrightarrow} (\lambda_{\mathbb{P}}\_.\mathbb{P}[\text{do\_log}])P[v] \\
&\overset{\text{Data Red Frame Rand}}{\Longrightarrow} \text{fail}
\end{aligned}$$

Trusted expressions may have side effects involving only trusted code, as long as there are no β-reductions into that trusted code through a guarded substitution.

The calculus we expose does a full fine grained dependency analysis of code, at each point tracking all the participants who have the ability to influence the expression by tracking all values in β-substitutions.

The requirement that the permissions check on an application be performed regardless of strictness in the operand allows us to use dummy operands as an indicator of permission to perform an application. Our *log* operator above was of the form $\lambda_{\mathbb{P}}\_.e$, being constructed as explicitly nonstrict in the otherwise irrelevant operand. It would be wrong to replace $(\lambda_{\mathbb{P}}\_.e)\tau$ with just e (or $\tau; e$) since we would lose the permissions check on the dummy operand τ. We may refer to such a dummy operand as a '*token*', passed merely for its security information and not for its value. This is very close to the capability model ([SSF99], [Den76] and many others) where a token of a particular type may be required as an argument to a function; this token carries the caller's capabilities (privileges) required for continuing the computation within the function.

It may be possible in any large system to find a piece of code which autonomously performs some privileged task without a token, code of the form $\mathbb{P}[e]$ for a closed expression e. Such a piece of code may be returned from a privileged object, for example, as a callback $\mathbb{P}[\lambda_{\mathbb{P}} x.\mathbb{P}[e]\,\mathrm{ok}]$ to notify of an event. The expression may be extended as $\lambda\_.\mathbb{P}[e]$ and passed into any frame, which may later invoke the callback. This callback now has no way of discovering who is responsible for evaluating it. Even if it were wrapped with a token argument as $\lambda_P x.\mathbb{P}[e]$, that redex could again be wrapped as $S[(\lambda_P x.\mathbb{P}[e])\,\mathrm{ok}]$ (with $P \subseteq S$) and passed on. The callback has no way to identify who is responsible for the call, and risks violating our original intentions in some significant respect.[61]

The current system also has the restriction that any guarded term be a closed λ-term. The onus is on the programmer to identify and guard all β-substitutions into a privileged expression. This is feasible in a system with clear distinctions between 'inside' and 'outside', such as the syscall interface to a Unix kernel, where the data passed to a syscall consists only of the registers passed to the trap and all data explicitly copied from userspace by the kernel. It becomes much harder in any system with dynamic scope or state where not all substitutions may be automatically identified. There also exists the issue of identifying side channels by which information may be passed to an expression. If, for example, the time at which an expression is evaluated influences the reduction in any way, the time must be somehow framed as an argument and tested by the guard.

Our next step will address these problems by building a dynamic context for the expression containing information about all principals having the ability to perform β-substitutions into an expression.

8.13 Explicit Context Passing Form

The explicit context passing form of a lambda expression is the key link between the data inspection calculus of section 8.12 and the stack inspection calculus of section 8.16. We demonstrate that the stack inspection calculus may be expressed within the mutator of a data inspection calculus, and show how the special case *grant* operator of the stack inspection calculus may be described in terms of the *untaint* operator of the data inspection calculus.

8.13.1 Construction

We required $\lambda_P x.\mathbb{P}[e]$ to be a closed term in order that we may trap all β-substitutions performed within the expression e. However, in $C[e]$, the lambda context $C[]$ contains every

[61] While the details are outside the scope of this presentation, there also exist situations where the ability to cause code to be executed, even in the absence of the ability to influence execution in any meaningful way, can introduce vulnerabilities into a system. Examples would include timing attacks, something abstracted out by the calculus, or calls when another part of the system referenced by the callback is in a known state, for example reading a vulnerable configuration file.

β-substitution applicable to e, along with the frame from the immediate principal performing each substitution.

This contextual information can be made available to the λ_P operator in the data inspection calculus of section 8.12 if we add a fresh token ε to the system for the sole purpose of accumulating these frames in dynamic scope. Rule *Data Red Frame Rator* from figure 29 shows that an operand ε substituted into a framed expression collects a frame of the immediate principal performing the substitution. ε will at every point in computation have a frame from each principal possibly responsible for substituting an untrusted w into a non-closed protected expression. Since we will pass ε through every frame in the context, it will collect a frame from every principal *capable* of performing a β-substitution in e. Thus we now have information about which additional principals *might* have influenced β-substitutions in e and we may inspect this information rather than inspecting only the specific arguments to a closed term.

This is the first of the modifications from a data inspection calculus into a stack inspection calculus, as described in figure 25.

8.13.2 Formalism

Definition 8.5 (Explicit Context Passing Form). An expression e may be converted into the *Explicit Context Passing Form of* e, $T(e)$ by the following recursively defined transformation:

$$\begin{aligned}
T(x) &= x \\
T(\text{fail}) &= \text{fail} \\
T(ef) &= T(e)T(f) \\
T(\lambda_P x.e) &= (\lambda_P \varepsilon x.T(e))\varepsilon \qquad (6) \\
T(R[e]) &= R[\lambda\varepsilon.T(e)]\varepsilon \qquad (7) \\
T(\text{untaint } R \text{ in } e) &= \text{untaint } R \text{ in } T(e) \\
T(\text{grant } R \text{ in } e) &= (\lambda\varepsilon.T(e))(\text{untaint } R \text{ in } \varepsilon) \qquad (8)
\end{aligned}$$

The operational semantics of this calculus remain as those in figure 29 from section 8.12. In every context, there is an additional fresh variable ε. Frames are added to ε by rules *Data Red Frame Rator* and *Data Red Appl* applied to the construct generated by rule 7, and are preserved by rule *Data Red Frame Rand.* Thus the set of frames on the ε in any expression is a copy (possibly with duplicates) of all the frames in the context. However, the outcome of no expression depends on ε save for in the possible failure of guarded substitutions and for passing it on (with more frames) to sub-redexes.

The key point of interest in this definition is rule 8, in which we define the grant operator in terms of the untaint operator, but first we must prove the computational properties of explicit context passing form.

Theorem 8.1 (Computational Equivalence of Explicit Context Passing Form). *An expression in Explicit Context Passing Form is equivalent to the original expression if frames around the context token ε are not checked. Formally, if rule 6 is replaced by*

$$T(\lambda_P x.e) \quad = \quad (\lambda\varepsilon.\lambda_P x.T(e))\varepsilon \tag{9}$$

then $T(e) \equiv e$.

Proof. We prove that each rule individually maintains computational equivalence, and thus by induction, $T(e)$ maintains computational equivalence other than for security checks.

Base cases: There are two redexes of length 1. The theorem holds trivially for both of them.

$$\begin{aligned} T(x) &\overset{\text{def}}{=} x \\ T(\text{fail}) &\overset{\text{def}}{=} \text{fail} \end{aligned}$$

Inductive cases: Let $T(e) \overset{\text{ind}}{=} e$ for any redex of length less than n.

$$\begin{array}{rcl} T(ef) \overset{\text{def}}{=} & T(e)T(f) & \overset{\text{ind}}{=} ef \\ T(R[e]) \overset{\text{def}}{=} & R[T(e)] & \overset{\text{ind}}{=} R[e] \\ T(\text{untaint } R \text{ in } e) \overset{\text{def}}{=} & \text{untaint } R \text{ in } T(e) & \overset{\text{ind}}{=} \text{untaint } R \text{ in } e \end{array}$$

Rule eta cases: The remaining cases, those which introduce a new substitution of ε into any expression, use rule η from the λ-calculus ([Bar84]).

$$M \equiv (\lambda x.M)y \text{ when } x \notin \text{fv}(M)$$

Note that ε is a fresh token, thus $\varepsilon \notin FV(e)$. Therefore:

$$\begin{aligned} T(\lambda_P x.e) &\overset{\text{rule 9}}{=} (\lambda\varepsilon.\lambda_P x.T(e))\varepsilon \\ &\overset{\text{ind}}{=} (\lambda\varepsilon.\lambda_P x.e)\varepsilon \\ &\overset{\text{rule }\eta}{=} \lambda_P x.e \end{aligned}$$

Also:

$$\begin{aligned} T(R[e]) &\overset{\text{def}}{=} R[\lambda\varepsilon.T(e)]\varepsilon \\ &\overset{\text{ind}}{=} R[\lambda\varepsilon.e]\varepsilon \\ &\overset{\text{Data Red Frame Rator}}{=} R[(\lambda\varepsilon.e)S[\varepsilon]] \\ &\overset{\text{rule }\eta}{=} R[e] \end{aligned}$$

And:

$$\begin{aligned} T(\text{grant } R \text{ in } e) &\overset{\text{def}}{=} (\lambda\varepsilon.T(e))(\text{untaint } R \text{ in } \varepsilon) \\ &\overset{\text{ind}}{=} (\lambda\varepsilon.e)(\text{untaint } R \text{ in } \varepsilon) \\ &\overset{\text{rule }\eta}{=} e \end{aligned}$$

This last may appear surprising. However rule *Data Red Grant* gives that

$$\text{grant } R \text{ in } e \rightarrow^{S}_{.} e$$

unconditionally. Our new definition of *grant* in terms of *untaint* has been considered temporarily in the light of rule 9, which discards the frames on the token ε. It is therefore expected that this will expand *grant* to an identity operation without rule 6, which checks these frames using a guarded substitution.

□

Theorem 8.2 (Frames on the Explicit Token). *The set of frames immediately surrounding any free instance of the token ε in any expression include (but are not necessarily equal to) the totality of frames in the context of that expression, with the exception of the immediate principal.*

The phrase "*immediately surrounding*" merits explanation: we may consider $R[\varepsilon]$ to be an unframed token ε in a context $R[]$. However, the frame $R[]$ is still an "*immediately surrounding*" frame, and is therefore included by this phrase in the definition.

Proof. By induction over the size of the context.

Base case: A free instance of ε with no context has no frames.

Inductive case: Assume the theorem holds for all $C[\varepsilon]$ for $C[]$ if size k. We must show that the theorem holds for all instances of ε in contexts of size $k+1$.

If the extension of the context does not introduce any new frames, then this holds trivially. For example, it holds trivially for $C[e\varepsilon]$ or $C[\text{untaint } R \text{ in } \varepsilon]$. The only construction which introduces a new frame into the context of ε is given by the right hand side of rule 7 of definition 8.5. In this case, in an immediate context S:

$$\begin{array}{rcl} T(R[e]) & \overset{\text{def}}{=} & R[\lambda\varepsilon.T(e)]\varepsilon \\ & \overset{\text{Data Red Frame Rator}}{\Longrightarrow} & R[(\lambda\varepsilon.T(e))S[\varepsilon]] \end{array}$$

Now, since all framed subexpressions within $T(e)$ must also be transformed by rule 7 of definition 8.5, any instances of ε within further subframes inside $T(e)$ must be bound by a new $\lambda\varepsilon.T(e')$. Thus the only free instances of ε in $T(e)$ are within the immediate frame R only.

□

Since rule *Data Red Appl* checks the immediate principal explicitly when performing a substitution, we may check every principal in the dynamic context by checking ε in a guarded substitution.

8.13.3 Consequences

The data security tag on the variable ε in any expression contains all the security information from the context of the expression. This context is the call chain, sometimes called the *stack*, and thus represents the combination of all those principals capable of performing substitutions within the expression. We will consider the reduction

$$((\lambda fv.fv)(\lambda x.e))v \to e[x := v]$$

as an example. The initial expression has been rewritten using the transformation $T()$ to include the token ε. We use rule *Data Ctx Frame* to allow us to reduce subterms and save notation, and a * indicates a repeated application of a rule.

Example 8.6.

$$\begin{array}{rcl} P[(\lambda\varepsilon fv.f\varepsilon v)]\varepsilon S[\lambda\varepsilon x.e]T[v] & \overset{\text{Data Red Frame Rator}}{\Longrightarrow} & P[(\lambda\varepsilon fv.f\varepsilon v)\varepsilon S[\lambda\varepsilon x.e]T[v]] \\ & \overset{\text{Data Red Frame Rand*}}{\Longrightarrow} & P[(\lambda\varepsilon fv.S[f]\varepsilon T[v])\varepsilon(\lambda\varepsilon x.e)v] \\ & \overset{\text{Data Red Appl*}}{\Longrightarrow} & P[SP[\lambda\varepsilon x.e]P[\varepsilon]TP[v]] \\ SP[\lambda\varepsilon x.e]P[\varepsilon]TP[v] & \overset{\text{Data Red Frame Rator in } \langle {}^{P}_{\cdot} \rangle[]}{\Longrightarrow} & S[P[\lambda\varepsilon x.e]PP[\varepsilon]PTP[v]] \\ P[\lambda\varepsilon x.e]PP[\varepsilon]PTP[v] & \overset{\text{Data Red Frame Rator in } \langle {}^{P\cap S}_{\cdot} \rangle[]}{\Longrightarrow} & P[(\lambda\varepsilon x.e)SPP[\varepsilon]SPTP[v]] \\ (\lambda\varepsilon x.e)SPP[\varepsilon]SPTP[v] & \overset{\text{Data Red Frame Rand* in } \langle {}^{P\cap S\cap P}_{\cdot} \rangle[]}{\Longrightarrow} & (\lambda\varepsilon x.e[\varepsilon := SPP[\varepsilon]][x := SPTP[x]])\varepsilon v \\ & \overset{\text{Data Red Appl* in } \langle {}^{P\cap S\cap P}_{\cdot} \rangle[]}{\Longrightarrow} & e[\varepsilon := SPPP[\varepsilon]][x := SPTPP[v]] \end{array}$$

We note that the frame S now appears around ε, although the frame S was not in the context of the bound occurrence of ε in the first redex in the first expression. Our data security semantics are stronger than pure stack security semantics ([FG02]) in that they keep track of the fact that the value in the frame S participated in the computation; S became a part of the context even though it merely framed a value in the first expression!

The token ε is not in the trusted computing base and any redex may substitute an alternative token for ε. This prevents a caller from identifying the previous context of the call and is equivalent to *indemnification of the caller*.[62] However, any fresh token will be framed by the immediate principal when it is substituted by rule *Data Red Appl*, and therefore a fresh ε is equivalent to *untaint S in* ε with the old ε, which filters all information except that about the immediate principal from the frames enclosing ε. We may use the filtering effect of *untaint R in* ε to mask information selectively about the callers. It is still necessary to use the *untaint* operation to perform a callee indemnification.

The dynamic context is written as the lower element D of context in $\langle {S \atop D} \rangle[]$ and $e \rightarrow^{S}_{D} e'$. It is computed as the intersection of all frames in the context of a lambda expression. We express this as a conditional rule for the purposes of our calculus:

$$\text{Token Ctx Frame} \qquad \frac{e \quad \rightarrow^{R}_{D\cap R} \quad e'}{R[e] \quad \rightarrow^{S}_{D} \quad R[e']}$$

This rule will appear in section 8.14 as "*Token Ctx Frame*" as the only non-exceptional case[63] under which dynamic context is modified.

8.14 Implicit Context Passing Calculus

In the explicit context passing system, the onus is on the programmer to pass the token ε. However, we showed that there is nothing to be gained by abusing the ability to substitute an alternative token for ε.

8.14.1 Construction

Since ε is a merely a token framed by the dynamic context and nothing else, we may remove it from our expressions and rewrite the model without the clumsy addition of a token if we annotate our expressions with dynamic context. This dynamic context represents the frames that would have been around the token ε, had it been present. In order to make this simplification, we must change the operational semantics of framed expressions to build the dynamic context. We return to the original data inspection model in section 8.12 and modify it to track dynamic context.

[62] Substituting an arbitrary non-fresh value for ε would be equivalent to spoofing the indirect callers.

[63] Excluding grant and untaint.

8.14.2 Formalism

Context is now written as $\langle {S \atop D} \rangle []$. Since the dynamic context is a single principal, we must be able to collapse multiple frames into a single principal. This is a case of joint responsibility, thus the resultant designation of responsibility may only have privileges which are held by both of the original principals. We achieve both this and an emulation of the inductive behaviour of $(\lambda_P \varepsilon . e) PT[\varepsilon]$ on multiple frames by using the meet operator on P and T to generate $(P \cap T)[]$ and, by induction over all frames on ε, D. D will be a full principal in any execution of the model, but for the purpose of performing subterm reductions or establishing congruences, we may only need to evaluate partial context and hence D will be considered as always partial.

Token Ctx Rand	$\dfrac{e_2 \rightarrow^S_D e'_2}{w_1 e_2 \rightarrow^S_D w_1 e'_2}$
Token Ctx Frame	$\dfrac{e \rightarrow^R_{D \cap R} e'}{R[e] \rightarrow^S_D R[e']}$
Token Ctx Grant	$\dfrac{e \rightarrow^S_{D \cup (R \cap S)} e'}{\text{grant } R \text{ in } e \rightarrow^S_D \text{grant } R \text{ in } e'}$
Token Red Grant	$\text{grant } R \text{ in } w \rightarrow^S_D w$
Token Ctx Untaint	$\dfrac{e \rightarrow^S_D e'}{\text{untaint } R \text{ in } e \rightarrow^S_D \text{untaint } R \text{ in } e'}$
Token Red Untaint Frame	$\text{untaint } R \text{ in } P[w] \rightarrow^S_D (P \cup (R \cap S))[\text{untaint } R \text{ in } w]$
Token Red Untaint Value	$\text{untaint } R \text{ in } v \rightarrow^S_D v$
Token Red Frame Rator	$R[w_1]w_2 \rightarrow^S_D R[w_1 S[w_2]]$
Token Red Frame Rand	$(\lambda_R x.e)P[w] \rightarrow^S_D \begin{cases} (\lambda_R x.e[x := P[x]])w & \text{if } R \subseteq P = \text{true} \\ \text{fail} & \text{if } R \subseteq P = \text{false} \\ \text{(not reducible)} & \text{if } R \subseteq P = \mho \end{cases}$
Token Red Appl	$(\lambda_R x.e)v \rightarrow^S_D \begin{cases} e[x := S[v]] & \text{if } R \subseteq D = \text{true} \\ \text{fail} & \text{if } R \subseteq D = \text{false} \\ \text{(not reducible)} & \text{if } R \subseteq D = \mho \end{cases}$

Figure 30: Token inspection operational semantics

The semantics in figure 30 are identical to those of the data inspection model in section 8.12 augmented by the explicit contextual token ε, which has now been merged into the dynamic context. The major changes are the introduction of the new rule *Token Ctx Grant* and the modifications to rules *Token Red Grant* and *Token Red Appl.*

- **Token Ctx Grant** allows the discarding of privileges held by the immediate principal from the previous dynamic context, and is equivalent to filtering the frames around the token ε, as would *untaint R in ε*. Thus *grant* is not really a new operator, but is equivalent to applying the *untaint* operation to a dynamically scoped token.

- **Token Red Grant** is not interesting. It may help to consider it as the base case of a moderately complex induction over the length of v where *Token Ctx Grant* is the step.

- **Token Red Appl** has been modified to inspect D instead of S. This is the topic of discussion below.

  We still frame the value v with S as we take it out of the immediate context S. Future research may consider what happens if D is used as a frame.

8.14.3 Consequences

As mentioned in section 8.11.2, [FG02] makes all reduction in the λ_{sec}-calculus conditional on $D \subseteq S$. Our calculus is unconditional and inspects only D. However, inspecting D in rule *Token Red Appl* is shown to be strictly more restrictive than inspecting S as an immediate corollary of the following theorem.

Theorem 8.3. *In each context $\langle {S \atop D} \rangle []$ in a token inspection calculus, $D \subseteq S$.*

Proof. By section 8.11.4, $D \subseteq S$ in the root principal. There are two places where we change context.

- Rule *Token Ctx Frame* allows evaluation in a context $\langle {R \atop D \cap R} \rangle []$, since $D \cap R \subseteq R$ is an immediate corollary of definition 8.3.
- Rule *Token Ctx Grant* allows modification in a context $\langle {S \atop D \cup (R \cap S)} \rangle []$. By the hypothesis of induction $D \subseteq S$. $D \cup (R \cap S) \subseteq S$ by a manipulation of the axioms.

By induction on the length of an expression, in each context $\langle {S \atop D} \rangle []$, $D \subseteq S$. Therefore our token inspection model is always at least as restrictive as the data inspection model in section 8.12.

□

The relation $D \subseteq S$ is a desirable property which we will aim to maintain in any calculus. The immediate principal is a part of the context, thus if we ever violate this relation we will be allowing the immediate principal to perform a guarded substitution using privileges greater than itself. This would be a direct violation of the principle of attenuation of privileges from section 2.4.2. Were $D \subseteq S$ a condition of reduction, as in [FG02], it must be verified at every stage of an execution, and this introduces undesirable overhead. We would also be unable to establish congruences in the calculus. We should be able to deduce an equivalent of theorem 8.3 from the operational semantics of any useful calculus.

D is satisfied by any principal who could have performed a free variable substitution in $\langle {S \atop D} \rangle [e]$. Therefore by testing that D satisfies P in the expression $(\lambda_P x.\mathbb{P}[e])w$, we test every such principal and we need no longer make the restriction that the expression e be a closed term. We have therefore addressed the concerns stated at the end of section 8.12.

This mechanism is considerably coarser than the model in section 8.12 since a frame in the context will now affect the security context of side expressions as well as expressions depending directly on the arguments given. However, the model is not yet so strict as to make the assumption that the reduction of every application depends on every operand by framing the reduct with frames from the operand.

8.15 Dependency Tracking Calculus

Tracking the dynamic context is not the panacea it appears to be. While we are now aware of every principal which might have caused a substitution in any given expression, we are not aware of every principal which might have affected the substituted value. We could track frames on substituted operands in the dynamic context. This is equivalent to the assumption that the operand always affects the evaluation of any expression into which it is substituted, that the operand is as much a participant in the evaluation as the operator.

8.15.1 Construction

The rule *Token Red Frame Rand* is designed to be a substitution of a value with the associated frames in case the result does not depend on the operand. If the value is discarded, the associated frames designating responsibility will also be discarded. If we assume dependence on the operand, then under the assumption above, the frames around the operand will influence the evaluation of all subterms of the application and hence form a part of the context of the expression. We may therefore construct a simpler ruleset. We preserve all rules from figure 30 except *Token Red Frame Rand*, to be replaced as in figure 31.

$$\boxed{\text{Deptrack Red Frame Rand} \qquad vR[w] \rightarrow_D^S R[S[v]w]}$$

Figure 31: Dependency-tracking specific operational semantics

There are two occasions in the rules presented in figure 30 where a security test is performed. We have just removed one of them. The remaining test, *Data Red Appl*, is a test only against the dynamic context; we never inspect the frames around values directly.

This model is now very similar to the *Call By Value Dependency Tracking* model presented in [FG02].

8.15.2 Consequences

We originally introduced dynamic context to cover the issue of requiring protected lambda terms to be closed. With the dependency tracking modifications to the calculus, any protected term may be an open term and security is guaranteed as it is in the data inspection calculus with closed terms. However, the protection system risks becoming excessively conservative and unworkable, since if an untrusted term has been used in a substitution at any point in the context, the application rules will return a fail.

8.16 Stack Inspection Calculus

The situation in section 8.15, where we consider only the dynamic scope in a security check, suggests that we consider an even simpler mechanism: that of stack security. Stack inspection is the last calculus on our journey from idealism to practicality, and is of interest because it is both the most widely used and the most vulnerable to attack.

8.16.1 Construction

The guiding principle of stack security is that only the substituting principals matter, not the substituted values, under the assumption that any principal will verify any value it substitutes. We model this by discarding the frames around all values and only maintaining the frames around applications in the context.

This is in violation of our requirement derived from definition 8.4 that we be able to identify all principals with the ability to affect the reduction of an expression e. However, we will present the model to show that it is a weakening of our previous models.

8.16.2 Formalism

We discard frames around values using a rule $P[w] \rightarrow^S_D w$. We consider the unframed value to be the "normal form" of the value and require all values to be reduced to an unframed form before substitution or application. For completeness and clarity, we represent the complete ruleset here. We remind the reader that v represents an *unframed* value.

Stack Red Frame	$R[v] \rightarrow^S_D v$
Stack Ctx Rand	$\dfrac{e_2 \quad \rightarrow^S_D \quad e_2'}{v_1 e_2 \quad \rightarrow^S_D \quad v_1 e_2'}$
Stack Ctx Frame	$\dfrac{e \quad \rightarrow^R_{D \cap R} \quad e'}{R[e] \quad \rightarrow^S_D \quad R[e']}$
Stack Ctx Grant	$\dfrac{e \quad \rightarrow^S_{D \cup (R \cap S)} \quad e'}{\text{grant } R \text{ in } e \quad \rightarrow^S_D \quad \text{grant } R \text{ in } e'}$
Stack Red Grant	$\text{grant } R \text{ in } v \rightarrow^S_D v$
Stack Red Untaint	$\text{untaint } R \text{ in } e \rightarrow^S_D e$
Stack Red Appl	$(\lambda_R x.e)v \rightarrow^S_D \begin{cases} e[x := v] & \text{if} \quad R \subseteq D = \text{true} \\ \text{fail} & \text{if} \quad R \subseteq D = \text{false} \\ \text{(not reducible)} & \text{if} \quad R \subseteq D = \mho \end{cases}$

Figure 32: Stack inspection operational semantics

The ruleset of figure 32 is very similar to that presented in [FG02], the major changes being the transformation of the check operator into the guarded-λ form and the addition of the *untaint* operation as a null operator for compatibility with previous models. The descriptions and justifications of most rules are similar to those after figure 30.

Since we have decided to ignore frames on values and preserve only those which form contexts in their own right, we are able to perform several simplifications in the calculus.

- The induction performed by *Token Red Untaint Frame* and *Token Red Untaint Value* becomes a *no-op* in *Stack Red Untaint* since frames will be discarded.
- Rules *Token Red Frame Rand* and *Token Red Frame Rator* are unnecessary since the frames will be discarded.

- Rule *Stack Red Appl* does not frame the substituted value with the immediate principal, since the frame will be discarded.

8.16.3 Consequences

The weaknesses of stack inspection security models are known ([FG02]), but are exhibited especially clearly in the light of definition 8.4, that we wish to identify all principals capable of affecting the reduction of an expression e. What we achieve by stack inspection is to identify all the principals capable of performing a β-substitution in e, but we fail to identify the principals capable of influencing those β-substitutions. It is therefore possible to return a Trojan value into an expression and perform a malicious operation.

Consider the following situation:

$$\begin{array}{rcl}
(P[\lambda f.f \text{ `secrets'}])(S[\lambda n.\text{foo } n]) & \overset{\text{Red Frame}}{\Longrightarrow} & (P[\lambda f.f \text{ `secrets'}])(\lambda n.\text{foo } n) \\
& \overset{\text{Red Frame}}{\Longrightarrow} & (\lambda f.f \text{ `secrets'})(\lambda n.\text{foo } n) \\
& \overset{\text{Red Appl}}{\Longrightarrow} & (\lambda n.\text{foo } n) \text{ `secrets'} \\
& \overset{\text{Red Appl}}{\Longrightarrow} & \text{foo `secrets'}
\end{array}$$

All the frame information is lost but we might still be performing a malicious act!

Stack inspection is considerably easier to implement than data inspection, since it requires little new data structure in the mutator. Most mutators already maintain a stack with pointers to code in each stack frame, by which responsible principals may be identified. This makes stack inspection popular in spite of its weaknesses. We will encounter stack inspection again in our case study of Java in section 9.5, and we will describe examples of data inspection in sections 9.9 and 9.11.

8.17 Consequences of Calculi for Identification of the Current Principal

8.17.1 Relationships between our Calculi

Our presentation of a calculus for a data inspection security model in section 8.12 and our calculus for a stack inspection security model in section 8.16 are orthogonal in the sense that they talk about largely non-overlapping concepts and structures. Each has an identity transformation in situations where the other would manipulate a structure. They may be combined into a common security calculus which satisfies all the requirements satisfied by each model individually; the result is the token passing model we used to derive the stack inspection model from the data inspection model. A token passing system with an explicit ε is a pure data inspection system which satisfies all the requirements of a stack security system, save that there is some onus on the programmer to maintain the token ε. Considered as a stack security mechanism, this model preserves frames from values either by substitution or in the partial context and hence may identify any principal which could influence the evaluation of an expression. The only significant issue is whether or not we consider a reduction always to depend on the operand.

This combination encourages us to consider that there may exist many possible calculi which satisfy some of our initial requirements and that they may be broken down into orthogonal structures and techniques. This is a topic for future research.

8.17.2 Partial Principals and Subterm Reduction

Our approach differs significantly from previous presentations of λ_{sec}-calculi in [PSS01] and [FG02] in the use of partial principals and hence partial contexts. [FG02] states explicitly that, "*We allow $e \rightarrow^S_D e'$ only when $D \subseteq S$*". [PSS01] is less clear but makes the same requirement. Both of these presentations calculate with full principals, and therefore reductions can only be made with full knowledge of the context of evaluation. Thus the calculus becomes merely an interpreter for the language. However, we would like to be able to identify contextual equivalences which may be reduced within any context $C[]$.

The ability to reduce terms without full knowledge of the context is extremely important; it transforms our reduction rules from an interpreter for a given computer program into a set of congruences which may be applied in general to construct staging transformations of programs. Our calculus does not have the requirement that $D \subseteq S$ be asserted at every step of the reduction. By theorem 8.3, we may then construct congruences and perform unconditional partial reductions of an expression.

In the case where a reduction may not be performed, an appeal may be made to a wider context under an appropriate *Ctx Frame* rule. The intersection of principals generated by this rule will create a dynamic context containing more information than the inner context previously considered. If appeal is always made to full context, then this approach becomes equivalent to that in previous presentations; this is, however, not necessary and does not accurately reflect the lazy algorithm used in actual implementations.

It is possible to prove with extra logic outside the scope of existing λ_{sec}-calculi that certain decisions may be made without full knowledge of the context, but we do not believe that this ability has yet been encapsulated within a λ_{sec}-calculus variant.

8.17.3 Lazy Evaluation with Partial Principals

Let the stack be a set of frames $f_i, i = 0 \ldots k$ counted from the root principal to the immediate principal. Each frame has associated with it the static principal of the frame, principal(f), and the set of privileges granted to the stack by that principal in that frame, grants(f).

The naïve routine for the identification of the current principal in a stack security system must identify a privilege which is the intersection of the privileges of every principal on the stack.

$$\text{current\_principal} = \bigcap_{i=0}^{k} \text{principal}(f_i)$$

Given an expression e which is to be reduced if the current dynamic principal has privileges satisfying R, we might use algorithm 3.

Algorithm 3 Constructive algorithm for evaluating satisfaction by the current principal

```
Require: An expression $e$ to be reduced if the current dynamic principal satisfies $P$.
  $p \leftarrow \emptyset$
  for each $f$ in stack do
    $p \leftarrow p \cup \text{grants}(f)$
    $p \leftarrow p \cap \text{principal}(f)$
  end for
  if $P \subseteq p$ then
    return $e$
  else
    fail
  end if
```

The algorithm builds information about the privileges of the dynamic principal starting from the root principal. Initially, we assume that the environment of execution gives a root principal with no privileges. For each frame on the stack, we add the grants from that frame to the dynamic principal and remove all privileges not held by the static principal of the frame from the dynamic principal. We note that since $\text{grants}(f) \subseteq \text{principal}(f)$, and $g \subseteq u \Rightarrow (p \cup g) \cap u = (p \cap u) \cup g$, it does not matter in which order we evaluate the grants and the static principal. The ordering above is conservative in case $\text{grants}(f) \subseteq \text{principal}(f)$ has not been verified at any stage. On reaching the end of the stack, the set p contains all privileges granted to the dynamic principal and held by all subsequent principals on the stack.

This routine adequately implements an evaluation of the dynamic principal and permissions test according to the calculi found in [FG02] and [PSS01] but it does not adequately model the dynamics of our calculus since it requires a full context of evaluation. Intuition leads us to construct a lazier model according to the following rules:

Working from the end of the stack (the immediate principal) to the root:

- A privilege granted by a frame does not need to be held by any frame closer to the root. In our previous algorithm, it would be re-added to the dynamic principal when the granting frame is reached.
- A privilege not yet granted and not held by a static principal will cause immediate failure.

A lazy implementation is demonstrated in algorithm 4. This lazy algorithm has the following advantages:

- It uses no storage beyond the initial size of P (frequently $|P| = 1$).
- It considers only those privileges relevant to the computation (those in P).

Algorithm 4 Lazy algorithm for evaluating satisfaction by the current principal

```
Require: An expression $e$ to be reduced if the current dynamic principal satisfies $P$.
  $p \leftarrow \emptyset$
  for each $f$ in stack do
    if $|P - \mathit{principal}(f)| > 0$ then
      fail
    end if
    $P \leftarrow P - \mathrm{grants}(f)$
    if $|P| = 0$ then
      return $e$
    end if
  end for
  fail
```

- It considers only those frames relevant to the computation (until all privileges in P have been granted).
- We do not have to assume that the principals themselves are countable, bounded or finite! We need only that $|P|$ is finite (in practice it usually has size 1), and that we have a membership operator for permissions such that we may implement the set difference. This membership operator is nothing special, it is $r \in [s, o]$.

This consideration of only that information which is relevant is modelled precisely by our tristate logic of partial principals. However, it is not modelled appropriately by previous calculi of [FG02] or [PSS01], and the correctness of this model is not directly provable in those calculi.

Unsurprisingly, implementations in the JVM ([Gon03]), the CLR ([Mic03b]) and other models use variants of this lazy algorithm, and Li Gong gives some further discussion in [Gon03] about the relative advantages of each algorithm.

8.17.4 Abstract Interpretation with Partial Principals

Our use of partial principals also allows us to create reductions conditional only on partial context, thereby establishing usable congruences. If a term establishes sufficient context that a subterm may be reduced within that context, we may perform reductions independent of the larger context of the term. This is the basis for correct optimising compilers. We use the data inspection model of section 8.12 to show that it is possible to optimise security systems at compile time without full knowledge of the context of any call.

The simplest possible case of a nontrivial optimisable sub-expression with security primitives is:

$$(\lambda_R x.e)(\mathrm{untaint}\ P\ \mathrm{in}\ f)$$

In this case, both R and P are constants, and hence full principals, so we may make the decision at compile time as to whether $R \subseteq P$ is true or false. If it is true, we may remove the security check and reduce the expression to $(\lambda x.e)f$, and if false, to *fail*. We may perform this reduction without first reducing the operand *untaint P in f* to a value w, since the *untaint* operator will ensure that $P \subseteq T$ for each frame T on w.

We derive the minimal nontrivial optimisable sub-expression for stack security from the ε transformation from definition 8.5. We will discuss only the case where $P \subseteq R$; all other cases reduce to *fail*.

$$\begin{aligned}
(\lambda_R \varepsilon x.e)(\text{untaint } P \text{ in } S[\varepsilon]) &\overset{\text{Data Red Untaint Frame}}{\Longrightarrow} (\lambda_R \varepsilon x.e)(S \cup P)[(\text{untaint } P \text{ in } \varepsilon)] \\
&\overset{\text{Data Red Untaint Value}}{\Longrightarrow} (\lambda_R \varepsilon x.e)(S \cup P)[\varepsilon] \\
&\overset{\text{Data Red Frame Rand}}{\Longrightarrow} (\lambda_R \varepsilon x.e[\varepsilon := (S \cup P)[\varepsilon]])\varepsilon \\
&\overset{\text{Data Red Appl}}{\Longrightarrow} (\lambda_R x.e[\varepsilon := (S \cup P)[\varepsilon]])
\end{aligned}$$

The result is to augment the frames on ε by the permissions in P and continue evaluating subterms of e in the context of this new ε. The dynamic context D represents the meet of frames S on our temporary token ε. By the distributive law on the lattice of partial principals, this is equivalent to augmenting D and evaluating subterms in the context of the augmented D, in other words, in the context provided by the rule *Stack Ctx Grant*:

$$\text{grant } P \text{ in } \lambda_R x.e$$

Thus we have derived the minimal optimisable sub-expression for a stack security model. This expression will reduce to $\lambda x.e$ if $R \subseteq P$ and *fail* otherwise.

We must have a machine which applies this data security model to all operations. Any elementary operation which combines two pieces of data to produce a result, however simple, must calculate the intersection of the privileges of the input data and attach that information to the result. This may create considerable overhead in any interpreter. However, we note that this extra computation may be optimised considerably using abstract interpretation at compile time, producing an expression for the privilege of the output of a block of code with respect to the privileges of the inputs, and without overloading every machine operation at runtime. This was discussed in [Ørb97].

8.17.5 Consequences of Enforced Security

As our protection model becomes more sophisticated and more powerful, so it also begins to enforce security in places where previously we might previously have opted for laxity. We will demonstrate this by the introduction of a new example, which we will call, "*Who maintains the compiler?*". However, the principle applies (with slight modification) to editors, filesystem utilities, almost any object which is shared between users and must access protected operations.

Example 8.7 (Who Maintains the Compiler?). Let Alice have privileges A and Bob have privileges B. Let the set $B - A$ be nonempty.

Let Alice be responsible for maintaining the compiler. In order to prevent trivial breaches of security, code giveaways are disallowed (section 2.4.2). Therefore the compiler object must have at most a privilege C which is a subset of Alice's privilege, $C \subseteq A$.

Let Bob use the compiler to generate Object. At the point when the object is created, the stack will look like (in the simplest case):

$$\mathbf{Bob} \rightarrow \mathbf{compiler} \rightarrow \mathit{create\_file}(\mathbf{Object})$$

The privileges of this stack will be $B \cap C$. But $C \subseteq A$ so $(B \cap C) \subseteq A$. At no point may any computation involving the Compiler exceed Alice's privilege, and therefore Bob may not use the compiler to construct an Object to perform any task which Alice does not have the privilege to perform. If $B - A$ is nonempty, as is usual, since Bob will usually have some privilege that Alice does not, then Bob cannot compile any object which uses all his privileges.

While this appears unnecessarily restrictive, Ken Thompson's famous compiler hack has entered into hacker legend for exploiting precisely this vulnerability by modifying the compiler to recognise when it was compiling the `login` binary and introducing a vulnerability that would allow him to log in to the system. The compiler would further recognise when it was recompiling itself and reintroduce the vulnerability into any new generations of the compiler. He presented the vulnerability in his 1983 Turing Award lecture to the ACM, a transcript of which may be found in [Tho84]; the hack is briefly described in [Ano].

The only principal which may maintain a system-wide compiler is the superuser. The same problem applies to almost all elements of a system which require privileges either to write or read files. Although we have improved security and simplified the task of constructing secure programs, we have also increased the hard requirement for effort on the part of the system administrators to maintain this security. It is no longer possible to make the compiler writable to some trusted third party because the system enforces constraints which will identify this as a vulnerability. Since most modern systems only allow the superuser to maintain elements of the system, this does not entail much change, but this policy is now enforced rather than optional.

8.18 Other Comparisons with Previous Work

8.18.1 Security Beyond Dynamic Context

It is the assumption of dependence on the operand that allows [PSS01] and [FG02] to move frames from the operand into the context and hence divide the λ_P operation into two operations: a traditional λ which performs unconditional β-substitution and an operation *check p for e* which performs a security check based only on the dynamic context. We do not make this transformation, primarily because we do not limit security tests to the dynamic context. *check* can be emulated in our model of section 8.15 using an expression of the form $\lambda_P\_.e$.

The *check* operator is itself defined as an abbreviation

$$\text{check } p \text{ for } e \mathrel{\widehat{=}} \text{test } \{p\} \ \mathbf{then} \ e \ \mathbf{else} \ f$$

The *test* operator performs two tasks: that of a permission test and that of a conditional. We may add an atomic conditional and a predicate *failp* to our language such that

$$\text{test } R \;\; \mathbf{then} \;\; e \;\; \mathbf{else} \;\; f \mapsto (\lambda x.\, \mathbf{if} \;\; \text{failp}(x) \;\; \mathbf{then} \;\; f \;\; \mathbf{else} \;\; x)((\lambda_{R\_}.e)\, \text{ok})$$

which performs the same task as the *check* operator. We consider the clause *else f* to be unnecessary in the model since it may be emulated by identifying the *fail* from any failed λ_P. Removing *else* allows us to perform the extension to our definition of λ_P, thus allowing guarded substitutions.

8.18.2 Nonfatal Exceptions

[FG02] defines *fail* as an *abnormal termination* and a corresponding rule

$$v\, \text{fail} \rightarrow^{S}_{D} \text{fail}$$

This models the behaviour of systems which throw a fatal exception on failure but fails to model those systems which do not consider failure of an operation to be a failure of the thread. We allow the modelling of all systems by removing this rule. We then consider the expression *fail* to be a discriminated expression similar to ℧ which will cause most operators passed *fail* as an operand to return *fail*.

Since both [PSS01] and [FG02] are based on the stack inspection model, they do not include the *untaint* operator, and the data inspection models in [FG02] do not include it. This weakness prevents an extension to a fine grained data inspection calculus, since *grant* is shown to be a special case of *untaint* in section 8.13.

8.19 Summary of Vulnerabilities

Our greater understanding of the way systems compute allows us easily to identify and classify vulnerabilities in existing systems. These vulnerabilities fall into three main categories:

1. **Untrusted Callee or Trojan Horse:** In the case where the principals responsible for the behaviour of a called piece of code are not accounted for in the calculation of the current principal, such a principal may create a malicious piece of code to be called with higher privileges than the principal itself holds.

2. **Untrusted Caller:** In the case where the principals responsible for calling a piece of code are not accounted for in the calculation of the current principal, a principal may call a trusted piece of code in undesirable circumstances or with unfortunate arguments.

3. **Untrusted (Returned) Value:** In the case where the principals responsible for generating a piece of data used in computation are not accounted for in the calculation of the current principal, a principal may create a malicious piece of data to influence a trusted computation in other ways.

This case is usually distinguished from case 2 by the fact that the malicious principal is no longer on the stack or taking any further active part in the computation at the time the computation is influenced. This is an essentially passive attack.

We now need refer to vulnerabilities only by name and classification, as in the case studies in section 9.

8.20 Summary

We have clearly identified and formally defined the concept of the current principal to the point where we are able to remove forever from the vocabulary of protection systems any informal use of the words "*trusted*" and "*untrusted*", replacing them with words such as "*responsible*".

We extended existing λ_{sec}-calculi in order to produce a correct calculus for identifying a current principal according to definition 2.11. Since, in practice, it is only necessary to know whether the current principal has a particular set of privileges, these λ_{sec}-calculi have been expressed in a logic of partial information.

We developed several related models in the λ_{sec}-calculus framework. The relationships are shown in the map in figure 25 on page 146, and we summarise the models themselves here.

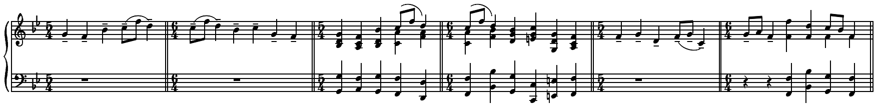
- **Data Inspection**
  - All responsible parties are identified.
  - *Protected terms must be closed.*
- **Explicit context passing**
  - Principals substituting values are tracked.
  - *Principals generating values are not tracked.*
  - *The programmer is responsible for the context token.*
- **Implicit Context Passing**
  - Principals substituting values are tracked.
  - *Dynamic context is automatically maintained.*
  - *Principals generating values are not tracked.*
- **Dependency Tracking**
  - All responsible principals are tracked.
  - *Extremely conservative.*
- **Stack Inspection**
  - Only depends on dynamic context.

  - *Violates responsibility by discarding frames.*
  - *Vulnerable to Trojan Horse values.*

Material which follows from this section may be found in:

- Section 9: With an understanding of the concepts and methods of computation of the current principal, it is extremely easy to classify systems by the mechanism they implement, and thus immediately understand the vulnerabilities in any existing system.
- Section 10: Given that we know which models are good and which are poor, we can now construct systems which are known and proven good for new application areas. However, there is a little more magic in the interaction between the privilege lattice and the transitive relations from section 7, all described in section 10.
- Section 12: The work in this section has led to the development of some new techniques for implementing the λ_{sec}-calculi, particularly the data inspection calculus of section 8.12. These techniques are introduced briefly in section 12.

But how does this apply to . . . ?



9 Case Studies

It is hard to be exhaustive in a study of mechanisms for the identification of the current principal since there are so many specifics of implementation and combinations of models from section 8 in use in the wild as to make a comprehensive classification impossible. We will instead study some implementations from well known operating system systems and cover most of the theoretical models from section 8.

As we perform each case study, we will also discuss the mechanisms used by the system under study for controlling access and for modifying the access specifications. Unfortunately most of the systems we will encounter in our case studies are of very simple forms, with notable exceptions, including EROS, CAP and Anarres II. It may seem peculiar that these most versatile systems are not in common use. We might propose many reasons for this, including economic pressure and the understanding of the mechanisms at the time. However, even in the case of these most advanced production systems, some of the important features of our *ideal* system from section 7.10 are missing. The feature most notable by its absence is the "*explanation*" or "*justification*" of access, as described in section 7.9. The only system described here which offers this capability is the author's own experimental system, the Anarres II library.

9.1 UID Based Mechanisms

9.1.1 Introduction to UIDs

Frequently, a choice of representation for principals is made before the mechanisms surrounding the various representations are fully understood. In such cases, a protection system must be constructed around the chosen representation. The UID is an example of such a representation, having evolved from the identifier for a protected memory space within a time sharing system such as CTSS ([Cri65], [SS75]), rather than being a purpose-designed representation. The UID is the simplest possible representation of a principal: an opaque identifier.

Since UIDs are opaque, no computation may be performed with them. This imposes many restrictions on which mechanisms may be used to identify a *current* principal. Any mechanism for the identification of the current principal using UIDs must focus around the choice of one particular UID as the responsible party, and hence the current principal.

Many of the problems with UIDs stem from the impossibility of combining two distinct UIDs into a new principal having privileges no more than the privileges of either, as required by section 8.5. It is usually impossible to evaluate the privileges of a UID: since the UID itself has no structure, UIDs are usually used in conjunction with an ACL[64] representation of the access matrix, and it is computationally intensive to invert ACLs in order to generate useful information about principals. Thus it is impractical, given a set of UIDs, to construct any useful partial ordering (such as that in definition 8.3) for use in any calculus of inspection. This choice of representation necessarily restricts us to the simpler and more vulnerable models. We will demonstrate that many of the theoretical problems suggested in sections 8.3 and 8.4 arise in real world systems using UIDs.

9.1.2 Construction of UID Based Mechanisms

Definition 9.1 (UID). A "*User Identifier* or *UID* is an opaque label identifying a principal.

There is a bijection between the UID and the conceptual principal which it identifies. Interestingly, our simplified definition of a UID is almost identical to the original definition of a subject in definition 4.1, and given the nature of the systems which were contemporary with [HRU76], it seems likely that the subject is a theoretical model of the UIDs used by systems at the time. The POSIX standard makes a slightly more specific definition of *UID* for POSIX systems; the specifics are entirely to do with storage; no computation is performed with UIDs, therefore the preceding definition will suffice.

Since UIDs are generally incomparable, and there is usually no discriminated UID having no privileges, there is no UID which has permissions less than those of each and every principal in the call chain of a computation. It is therefore only possible to combine the responsibility of any two principals by selecting one of them as the current principal. As we might remember from the introduction to section 8.2, this is rarely if ever the correct thing to do.

The possible mechanisms vary only in the choice of principal from the call chain. The only principals immediately identifiable for a distinct role in the computation are the root principal and the immediate principal. However, we must also consider the consequences of using some non-immediate principal, or using the root principal for some cases and the immediate principal for others. Luckily, an extremely well understood and readily available example of all these mechanisms is available to us for study.

[64] See section 2.3.4.

9.2 Case Study: Unix (1970)

> ... the number of UNIX installations has grown to 10, with more expected ...
>
> — Dennis Ritchie and Ken Thompson, June 1972

The first versions of Unix were written in BCPL, and an early version of the "`UNIX PROGRAMMER'S MANUAL`" by K. Thompson and D. M. Ritchie is dated "November 3, 1971". This document already includes manual pages for such familiar commands as `chmod` (change file permissions) and `chown` (change file owner). In 1972, the system was rewritten in Ritchie's new language, C. In 1974, the system was publicly unveiled, and after a July publication in the Communications of the ACM, demand for the system grew.

For a number of years, the evolution of Unix was continued by hackers mailing source code to one another, fixing bugs as time went on. When Berkeley Software Design, Incorporated (BSDI) was formed to provide commercial support for this code, AT&T, the then-owners of Unix, mailed them a lawsuit. AT&T lost, there was a settlement, and free Unix was born. ([McK99], [Rit79], [Tec02], [Mof03])

It would be an understatement to say that modern Unix is a well known system. The POSIX standard (IEEE 1003.1, [PASC97]), the Single Unix Specification ([OG03]) and the advent of open source Unixes including BSD and Linux have made the mechanisms by which it operates especially transparent and well understood. Unix makes use of UIDs for computations of security, and is an ideal candidate for a case study of a UID based system.

9.2.1 Identification of the Current Principal

There is a discriminated UID representing the superuser; this UID automatically has all permissions as a special case.[65] We will call this discriminated superuser UID `root`. All other UIDs are called *users* and are incomparable.

Unix assigns a UID to every process. Since the rules for the assignment of UIDs to processes are so restrictive,[66] it is sufficient to consider that each non-`root` process has a fixed UID. Normally, we would consider the processes to be the principals of the system since they are the actors within the system. However, since processes with the same UID may be freely created and destroyed, two processes operating under the same UID are indistinguishable for all purposes relating to security. For the purposes of analysis, it is therefore of little importance whether we consider the principals of the system to be the processes or the UIDs.

The Unix kernel evaluates the current principal by inspecting UID of the current process (or '*current task*' in some kernel terminology), the '*immediate principal*' in our terminology. Understanding the assignment of UIDs to processes is therefore the key to understanding the algorithm for identifying the current principal. The call chain of processes is extended by using `fork` and `exec` for loading new binaries (and possibly changing privileges), `setuid` and friends (for changing privileges) and any Inter-Process Communication (IPC). For the purposes of

[65] We will ignore the complications of more recent capability extensions since these are not widely used and are not core to the operation of the system.

[66] Only the a superuser process is allowed to change its UID, by doing which, it gains nothing.

analysis, `fork` itself is not of interest since it introduces neither new code nor new privileges. An `exec` without a `fork` replaces the last principal on the call chain instead of adding a new principal to the chain; the security impact is identical since in both cases, the intent of the calling principal, the previous immediate principal, is preserved. We will consider the interesting system calls case by case.

1. **A new process may inherit its UID from its caller, or 'parent', the process calling `exec`.** (`exec` of a normal binary)

   This is equivalent to the root principal model in section 8.4.

   For this to be correct, we must assume that the parent process trusts the author of the object code, since the verification of code is immediately the halting problem and thus the intent of the called program cannot be verified.[67] The only method to ensure safety is to restrict executed code within the object to that from trusted sources. It is a nontrivial problem to identify the source of any program, a problem which is roughly equivalent to the problem of identification of the current principal. Traditionally, exploits of this mismatch of intent are called "*Trojan Horses*", and they will abound in a UID system with little or no possibility of detection or protection.

Vulnerabilities: Vulnerability 1 in section 8.19 (the untrusted callee) applies. The new process executes with the privileges of the caller, and the author of that process has liberty to code maliciously as he will.

2. **A new process may inherit its UID from its author.**[68] (`exec` of a SUID binary)

   This is equivalent to the immediate principal model in section 8.3.

   For this to be correct, we must assume that the called object trusts or can verify that no data passed to it will lead to a breach of security. This involves an analysis of the security implications of every piece of data passed to the process. This is not always possible, although many programs rely on being able to do this, including `Xwrapper` from the XFree86 project, `wrapper` from Majordomo, and others.

Vulnerabilities: Vulnerability 2 in section 8.19 (the untrusted caller) applies. Constructing buffer overruns, finding unchecked execution paths in SUID binaries and other related tasks are traditional hobbies for bored Unix hackers.

3. **A running process will keep its existing UID if called by IPC.** (Any IPC call)

   This is equivalent to the immediate principal model in section 8.3.

   Data may be passed from process to process by any IPC mechanism, and used in computation. This assumes that the process trusts or can verify the safety of any data it will use in computation.

[67] See appendix A.5.

[68] In a more general system, a process might assign its own UID, subject to the restrictions of the principle of attenuation of privileges ([Min78]).

Vulnerabilities: Vulnerability 2 in section 8.19 (the untrusted caller) applies. Cases too numerous to list include X11 servers, NFS servers and an increasingly amusing number of bugs in Microsoft network servers.

We note that a thread of execution entering a daemon executing under a different UID (or indeed the kernel) is equivalent to executing an object which sets its own UID. The existence of the instance of the called object prior to its inclusion in the call chain does not affect the model. Therefore, this case is in fact no different from the previous case.

4. **An already running process may change its UID.** (`setuid` and friends)

   The designers of Unix were sufficiently paranoid about this feature to restrict its use almost completely from everybody but `root`, the Unix superuser ([OG03], [TCOS90]) and hence it has almost no impact on security. It will be ignored henceforth.

A pure parent-UID model with no SUID concept and no IPC is equivalent to the root principal model described in section 8.4 under which processes have no UID and the current principal is the current user. The distinction between assigning a UID to each process and using the root principal's UID is only significant when there are special case assignments of UID, for example SUID programs under Unix or IPC calls. In fact, the UID mechanism described above is a combination of the root principal model from section 8.4 and the immediate principal model from section 8.3 and inherits the vulnerabilities from both models.

When any process terminates, Unix returns a few bits of information to the parent process: an 8 bit return code and some information regarding which signal caused the process to terminate. Return codes are usually 0 for success, and some arbitrary integer for a failure. While Berkeley did attempt to standardise return codes for system programs, the idea did not become popular and so no computation is performed using return codes. Vulnerability 3 in section 8.19 (the untrusted value) rarely if ever applies to Unix systems.

9.2.2 Access to Protected Objects

Protected objects under Unix include files, message queues and shared memory segments. Each protected object has associated with it an owner UID, an owner GID, and a mode. The mode relating to protection consists of 11 bits, representing the read, write and execute permissions to the object for the owner UID, members of the group of users represented by the owner GID, and all other users respectively, and two set-id bits. These permissions are frequently written as `rwxr-xr-x`, to represent that processes running under the owner UID may read, write and execute the file, and all other UIDs may only read and execute the file. If a file if SUID and executable, then the mode will be written as `rwsr-xr-x`.

The execute bit is largely advisory since most executables are readable. It is usually trivial for any user to create a copy of a file which he may then execute, although such a copy will be owned by the UID creating the copy and any SUID information will be lost. The execute bit is largely used to notify the system about which files were intended to be executed. Some system administrators will create custom non-readable SUID executables (mode `rws--x--x`) as an added security measure to prevent hackers from inspecting the binary of a program to find vulnerabilities.

The superuser `root` may access all objects, as a special case.

9.2.3 Modification of Rights

The Unix system is extremely restrictive. Only the superuser may create new UIDs and assign UIDs to groups. Under most modern Unixes, only the superuser may change the ownership of files to an arbitrary UID.

Any user may change the permissions or mode of a file, and may under certain circumstances change the ownership of a file to their own UID or change the group ownership of a file to a group of which he is a member.

These rules are deliberately designed to prevent the propagation of rights; no user may create any UID or any object under another UID. A user may only create more objects under his own UID, and as previously discussed, all such objects are indistinguishable from the perspective of security.

9.2.4 Specifics of Implementation

There is no 'bottom' principal with no privileges under Unix.[69] Frequently a system will create a `nobody` user for use as a principal with no privileges, but this principal is not differentiated from any other user non-root principal. It is simply another UID and holds principal as would any other. This occasionally causes security problems when two processes running under this `nobody` user are caused to interact by a malicious user.

Unix is actually a bounded system (see section 6.1); many implementations are limited to fewer than 65536 subjects. Safety is therefore decidable in very large time by a search algorithm. Also, under most installations, there is no command which allows a Unix user to create a new UID, thus in a formalism of Unix, it is impossible for any subject except root to create a new subject.

In order to extend to allow non-`root` UIDs to access parts of the system traditionally accessible only by `root`, it is usual to construct a process running as `root` (either from a SUID file or as a system invoked daemon) as a proxy for the system call and have that process check the data being passed it (in some arbitrarily complex manner) before passing this on to the system call. Rules for such checks can be very poorly defined. For example, in the case of syslog, the privilege required to write to a file in `/var/log` is usually `root`, but a message may be generated by a userspace program. The data is passed by IPC to a daemon with a UID of `root` which must check the data before using it in execution.

Incidental changes of UID may also occur when the change of UID is not deliberate but more of a matter of fact in the way the system is set up. This might, for example, occur when executing a CGI script in a web server on a system to which the user already has access. The web server frequently runs under a special UID '`httpd`', allowing the user to pass data and

[69] See section 7.9.8.

sometimes execute scripts as this UID but without deliberately benefiting from the change of UID. In the case of CGI script execution, this even can be a nuisance because the script may no longer be able to access the user files it requires to run. This case is not common, and designedly so.

This delegation of responsibility and ad hoc validation leads to the swiss cheese effect on security in systems using UIDs (see section 8.8).

9.2.5 Summary of Unix

Were Unix to be correct, its extreme restrictiveness would probably cause it to be decidably secure. However, the potential vulnerabilities in Unix and Unix-like systems are well known and legion. Unix attempts to balance the problem of granting sufficient privilege for the task with the problem of controlling the use of privilege; however the underlying structure does not permit both of these ends to be simultaneously achieved. Careful application of this model can make a secure and usable, if restrictive system, but it should not be a basis for the design of new security systems.

More recent generations of the Linux kernel ([KO04]) include alternative security models.

9.3 Summary of UID Based Mechanisms

All UID based mechanisms are incorrect and extremely vulnerable to attack by any party involved in the computation. The classic Trojan Horse example is the oldest demonstration of this problem, where an untrusted principal (the Trojans) entered into a computation with what was considered to be a trusted principal (the horse), with disastrous consequences. We must be able to include all principals in our computation of a responsible '*current principal*' and thus we must turn our attention to models which allow the combination of principals.

As a result of our analysis of mechanisms, we realise that a more complex representation of a principal is required. In order to implement any of the security calculi of section 8, we must be able to combine arbitrary principals into a possibly new principal with permissions not greater than any of the original principals, as described in section 8.5. The simplest possible representation of a principal which allows this combination is a set of privileges, the capability map from section 2.3.7. Given this representation, we may create a new principal with any set of privileges without being required to modify each and every ACL in the system.

9.4 Stack Inspection Based Mechanisms

Stack inspection is considerably more flexible than the UID based mechanism described in section 9.1 since it is the first of our systems to represent principals as privilege sets rather than as opaque objects. This flexibility allows the creation of principals with arbitrary privilege sets, and thus we may create a principal holding only those privileges held by each and every principal involved in a computation.

While this flexibility allows computation with principals, we must still be able to identify all principals involved in a computation, thus to combine their privileges into a current principal.

The stack inspection security model is only an approximation of this, focusing entirely on the call chain and neglecting to consider the effect of returned values. Even with this omission, the Java implementation is still a powerful mechanism since the simplicity of the returned values reduces their potential as bearers of malicious intent.

9.5 Case Study: Java (1991)

> It's surprisingly difficult to find a good name for a programming language, as the team discovered after many hours of brainstorming. Finally, inspiration struck one day during a trip to the local coffee shop.
>
> — James Gosling

> The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.
>
> — Donald Knuth[70]

9.5.1 Origins of Java

In the late 1970s, Bill Joy, one of the founders of Sun Microsystems, conceived of a new programming language which was to combine the best features of MESA and C. However, it was not until September 5th, 1990, that the "Green Project" was created to develop this language, then called "Oak", with a mandate to "do fewer things better". The focus at the time was on set top boxes, intelligent white goods and consumer electronics. In 1992, the first such device was ready. It was the "*7", a cross between a PDA and a remote control. Without a market, the project was rolled back into Sun in 1994.

It was realised that the design requirements for the uses of Java in set top boxes and consumer electronics were the same as the requirements for the world wide web: small, platform independent, secure and reliable code. The team began development from the newly available Mosaic browser into what became known as the HotJava browser. By March 1995, the team were preparing to release version 1.0a2 of the Java source code on the internet. On Thursday March 22nd, 1995, the San Jose Mercury ran a front page article four days earlier than expected, extolling the merits of Java. Suddenly the world took notice. But Java still had no practical place in the traditional Unix computing world until Marc Andreessen of Netscape joined Sun executives during a SunWorld keynote speech to announce that Netscape and Sun had signed a deal to provide Java technology for the now omnipresent Netscape browser. (Information from [Byo98] and [Har97].)

9.5.2 Evolution of an Architecture for Protection

Security has formed an integral part of the Java architecture since the very early days with the introduction of the abstract class `java.lang.SecurityManager` in JDK 1.0. The purpose of the

[70] Much to do with names, but nothing to do with Java!

original SecurityManager was to control access to system resources according to rules provided by a concrete implementation of SecurityManager. This SecurityManager implementation had access to some primitive native methods for inspecting the VM stack, but no standard representation for permissions or principals was available. Security was frequently implemented by simpler interactions between the SecurityManager and a trusted java.lang.ClassLoader object to prevent the loading of untrusted code into the virtual machine. Many earlier Java security models are explained diagrammatically in [Gon03]. (Information from [Fla96], [Sun96] and [Gon03].)

A new and more extensible security architecture for the system was designed for the release of Java 2, beginning with JDK version 1.2. The java.security.* hierarchy was introduced, with a security core being built on new classes within this hierarchy including Permission, ProtectionDomain, AccessController and AccessControlContext. Minor changes have been made to this architecture for version 1.4.2, the current version as of this writing, and our work will focus on this version. (Information from [Fla99], [Gon03], [Sun03a] and [Sun03b].)

9.5.3 Construction and Representation of Principals and Permissions

The concept of a principal as an entity with control over the behaviour of code within a system is represented in the Java model by the class java.security.CodeSource.[71] The CodeSource is a pair (*location*, *certificates*), where *location* is a location from which code may be obtained, and *certificates* is a list of the certificates of the authors of the code. This pair identifies the origin of the code, and the CodeSource has no structure beyond this. A concrete instance of java.lang.ClassLoader, frequently a subclass of of java.security.SecureClassLoader, is responsible for obtaining code from a given CodeSource. This SecureClassLoader will then extend the CodeSource by associating it in a one-to-one relationship with a java.security.-ProtectionDomain object and associate the new code with that ProtectionDomain.

The ProtectionDomain is the principal with which we may perform computation since it holds a representation of a set of permissions. It is the responsibility of the system administrator to assign privileges (java.security.Permission objects) to CodeSources, and thus to the ProtectionDomains associated with them, by defining a security '*policy*' (a global object which subclasses java.security.Policy).[72] There may exist many Policy objects in the system at any one time; they are interchangeable by any code with permission to do so. The existence of the Policy object is merely a convenience to permit this interchange; it is sufficient to ignore policy entirely and consider the ProtectionDomain as a principal with privileges.

[71] The Java 2 API specifies an interface java.security.Principal. This is a red herring and does not map to the logical concept of a principal as we have been describing it.

[72] By default, an instance of sun.security.provider.PolicyFile is used, but this may be configured by supplying an alternative class name as a value for the policy.provider property. [Gon03] states "*It is perhaps worth emphasising that such an override of the policy class is a temporary solution and a more comprehensive policy API will probably make this unnecessary.*"

9.5.4 Identification of the Current Principal

The Java 2 Security Specification ([Gon03]) identifies the two possible algorithms for evaluation of the current principal in dynamic scope, as suggested in section 8.17.3. It states:

> There are obviously two implementation strategies:
>
> - In an 'eager evaluation' implementation, whenever a thread enters a new protection domain or exits from one, the set of effective permissions is updated dynamically.
>
>   The benefit is that checking whether a permission is allowed is simplified and can be faster in many cases. The disadvantage is that, because permission checking occurs much less frequently than cross-domain calls, a large percentage of permission updates are likely to be useless effort.
>
> - In a 'lazy evaluation' implementation, whenever permission checking is requested, the thread state (as reflected by the current state, including the current thread's call stack or its equivalent) is examined and a decision is reached to either deny or grant the particular access requested.
>
>   One potential downside of this approach is performance penalty at permission checking time, although this penalty would have been incurred anyway in the 'eager evaluation' approach (albeit at earlier times and spread out among each cross-domain call). Our implementation so far has yielded acceptable performance, so we feel that lazy evaluation is the most economical approach overall.

This lazy evaluation method never actually computes a current principal. Instead, it is sufficient to identify whether the hypothetical current principal holds a given permission, and this is what we compute.

Interestingly, we noted in the summary of section 8.17.3 that if we used a lazy evaluation strategy, it was sufficient to have a membership operator for a permission and a principal, to ask, "*Does this principal have this permission?*" Permissions need not then be an enumerable set. Java takes full advantage of this possibility; the permissions in Java are not necessarily an enumerable set. Instead, a permission is represented by a `java.security.Permission` object with a method `bool implies(Permission p)`, returning **true** if the permission is a greater permission than `p` (the permission held implies the possession of the permission `p`) and **false** otherwise.

A permissions check in Java proceeds as follows:

1. A request for a resource is trapped before it enters native code.
2. If the request is a part of a backwards compatible API from Java 1, then the request is passed to the system security manager. The default system security manager constructs an instance of the `Permission` required to access the resource and continues with step 4.

3. Otherwise, the object trapping the resource constructs an instance of the `Permission` required to access the resource.

4. The static method

   `static void AccessController.checkPermission(Permission p)`

   is called. This method must evaluate whether the current principal holds permission `p`. If so, the method will eventually return and the request will proceed normally. Otherwise, it will throw an exception. The following steps describe this mechanism.

5. The `AccessController` constructs a `java.security.AccessControlContext` object, a representation of the stack, the dynamic context of the request. This context need only be an unordered set of principals in the dynamic context, in fact, a set of `ProtectionDomain` objects.

6. The `AccessController` calls

   `void acc.checkPermission(Permission p)`

   on the constructed `AccessControlContext` object, passing on the question asked in step 4.

7. For each `ProtectionDomain` in the `AccessControlContext`, the `AccessControlContext` calls the method

   `bool domain.implies(Permission p)`

   on the domain object. If a `ProtectionDomain` does not have the permission, a `java.security.AccessControlException` is thrown, to satisfy the requirement of step 4.

8. Each `ProtectionDomain` has a number of `java.security.PermissionCollection` objects. It identifies the collection which would contain the requested permission and calls

   `bool collection.implies(Permission p)`

   on that collection.

9. The `PermissionCollection` is a homogenous collection of `Permission` objects, any one of which might satisfy the given permission `p`. The `PermissionCollection` therefore calls

   `bool permission.implies(Permission p)`

   on each permission. If no permission returns **true**, then **false** is returned, and the `AccessControlContext` of step 7 will throw an exception.

9.5.5 Modification of Rights

We have remained fairly specific to the default implementations of the Sun Microsystems Java 2 Standard Edition. However, a study of the Java virtual machine specification in [LY97] or [MD97] will find scant mention of security, and certainly no description of the model above. Apart from some semantics of classloaders and the ability to inspect the virtual machine runtime stack, the virtual machine plays almost no role in the security model for the system. The security model is implemented almost entirely in the standard library and in native Java, with a little assistance from the virtual machine for identifying dynamic context. Descriptions of the security model are only to be found in [Sun03a] and [Gon03].

It is possible to write a Java Virtual Machine Standard ([LY97]) compliant virtual machine with no mention of security whatsoever; examples include the IBM Jikes RVM ([IBM03]), the Intel Open Research Platform ([IG03]) and the Aegis VM ([F$^+$03]). Most of these open source virtual machines run the GNU Classpath library ([GNU03]), an open source replacement for the Sun Microsystems Java class library. This library is also without security primitives, but this library is not a compliant implementation of the Java standard.

In the default Sun Microsystems implementation of the standard library, the possession of permissions within the virtual machine is controlled by an instance of a concrete subclass of `java.security.Policy`. A policy specifies which `Permission`s are to be granted to each `CodeSource`, and thus to each `ProtectionDomain`. The default `Policy` implementation provided by Sun Microsystems in their JDK requires that the virtual machine administrator write a policy file to be loaded by the system at boot time. In the default implementation, therefore, permissions are static and may be neither granted nor revoked.

Given the immense flexibility built into this standard library, it is impossible to give a set of hard and fast rules for modification of rights. It is possible to implement any protection system described in this paper within the JVM. The library itself provides a number of opportunities to customise the system and we make a note of some of them here.

- The system `SecurityManager`, by default loaded from the class `java.lang.SecurityManager`, may be replaced by specifying a different security manager class at system boot-time. Since, for legacy compatibility, all calls from standard Java classes call the `SecurityManager`, the `SecurityManager` is at liberty to make arbitrary decisions of security using the primitives available. This approach is deprecated in Java 2. This replacement is restricted to the administrator with the ability to specify properties at machine boot time.

- The system `Policy` object, by default loaded from the class `sun.security.provider.PolicyFile`, may be replaced, thus to specify a different allocation of `Permission`s to `ProtectionDomain`s. The class implementing `Policy` may be specified by the administrator at machine boot time, and the default `Policy` also specifies a permission restricting the replacement of the policy at runtime.

- A custom `ClassLoader` may associate an instance of any subclass of `ProtectionDomain` with a given `CodeSource`. The user-provided `implies` method on this `ProtectionDomain` may make arbitrary decisions about whether the domain implies a particular privilege. This association of `ProtectionDomain`s with `CodeSource`s is restricted by the ability to create `ClassLoader` objects, something controlled by a permission in the default implementation.

- The `implies` methods on user-provided `PermissionCollection` objects may make arbitrary decisions. No standard system permission will reside in a user-provided `Permission-`

Collection, and thus this may not be used to defeat the access control on standard resources.

- The implies methods on user-provided Permission objects may make arbitrary decisions. **This approach is recommended by Sun Microsystems** ([Gon03]), and this may easily be used to implement any of the models described in this paper. User provided permission objects may not supersede system permission objects, but must exist in a parallel namespace. This is partly ensured by a well behaved ClassLoader and partly by the license agreement with Sun Microsystems which states that no user will place objects into certain namespaces.

There is a downside to the current Java security architecture: While it is easy to extend the system with new rights and test for these new rights, it is almost impossible to provide new ways of accessing existing rights, for example to gain access to a FilePermission using a transitive relation such as that in section 7.9. This makes it almost impossible to do a meaningful manipulation of rights at runtime. The only way to implement our ideal protection system in the current Java architecture would require a replacement of the policy provider, on the assumption that the return values of Policy.getPolicy() and Policy.getPermissions() are not cached.

9.5.6 Specifics of Implementation

The AccessControlContext specifies which principals are involved in the computation, but there are many cases where this set is not equivalent to the set of ProtectionDomains in dynamic scope. We note a few such cases:

- The virtual machine and standard library jointly implement a

  AccessController.doPrivileged(PrivilegedAction action)

  which is the equivalent of the *grant* operation from the stack inspection calculus of section 8.16. The AccessControlContext must list only principals in the call chain *after* the doPrivileged call.

  A variant of this doPrivileged call also allows the inclusion of an arbitrary AccessControlContext in the dynamic context of the privileged call.

- The virtual machine supports asynchronous callbacks into Java code. The AccessControlContext for a callback must include the set of principals involved in the establishment of the callback as well as those in the call chain of the callback itself. There is some manipulation of AccessControlContext objects in java.lang.Thread to facilitate this.

- A ClassLoader operating in an untrusted ProtectionDomain, such as java.net.URLClassLoader, may remember the AccessControlContext of its own instantiation and include that context in the context of any privileged calls made.

9.5.7 Vulnerabilities

Stack inspection for the identification of the current principal is described in section 8.6 and is modelled directly by our Stack Inspection Calculus in section 8.16 with modifications as described in section 8.17.3. We are therefore extremely familiar with the vulnerabilities of the design.

Java suffers from vulnerabilities related to returned values, as does any security system based on stack inspection. Usually, a returned value can have little malicious impact, but in the case of Java, data can represent code, if passed to a classloader. Considerable effort is expended in the careful handling of bytecodes returned from remote servers. Certificates are checked, new principals and protection domains are created, and there is much wailing and gnashing of teeth.

All in all, the designers of Java have done a good job in mitigating the risks of this particular vulnerability in many cases throughout the standard library, and have thus made a significant step towards the objective of designing a system within which security vulnerabilities cannot arise. However, the potential for vulnerability remains.

Vulnerabilities: Vulnerability 3 in section 8.19 (the untrusted value) applies in exactly the form demonstrated in example 8.3 on page 139. Values returned to the thread may be constructed by objects which do not remain on the stack and so are not considered to form a part of the current principal. Such values must be checked very carefully before being handled in any security sensitive way.

We note that while the parent process has no way of verifying the behaviour of the child, the child process frequently has a way of verifying the (primitive) arguments passed to it. Therefore the model allows for the child process to raise the privileges of the execution thread (with dynamic scope) to any privilege at most that held by the process itself (according to section 2.4.2). This is the *grant* operation. In the presence of grants:

Vulnerabilities: If this feature is used, then vulnerability 2 in section 8.19 (the untrusted caller) and vulnerability 3 in section 8.19 (the untrusted value) will both apply. A child process performing a *grant* must first identify and verify all data and code which may affect privileged operations within the dynamic scope of the *grant* operation.

Since it requires special effort on the part of the programmer to raise the privilege of the thread, the programmer is assumed to be aware that he has raised the privilege and therefore to have taken special action to ensure that the data he has passed into the higher privilege thread is not going to cause a breach of security. Therefore vulnerability 2 in section 8.19 (the untrusted caller) only applies in the case of an explicitly careless programmer. The main vulnerability of the stack inspection security model remains vulnerability 3 in section 8.19 (the untrusted value) since there is no way of verifying the origin of a value in this model.

9.5.8 Summary of Java

The stack inspection model is the first of our practical models to have the advantage that consideration of security is not important to the average programmer. The ordinary mechanisms of the system ensure that the privilege of the thread does not exceed that of any of the *major* contributors to the intent of the thread and thus significantly reduces the load placed by considerations of security on the regular programmer. Verification of returned values is still important, but such vulnerabilities are rare and hard to exploit.

However, the Java model, while correct, is not dynamic: It is difficult or impossible to grant and revoke rights. This is more an accident of the implementation than a feature of the design, and [Gon03] promises a more comprehensive policy API in the future.

9.6 Case Study: .NET and the Common Language Runtime (1999)

> What is a .NET experience?
>
> The Microsoft .NET experience is a dramatically more personal, integrated computing experience using connected Web services delivered through smart devices.
>
> — Microsoft .NET FAQ

The Microsoft .NET framework is also rumoured to use a stack inspection based security model ([FG02]), but in the absence of either source code or any good documentation, it is very difficult to verify this. We provide this section to rectify what would otherwise be a glaring omission from this thesis, but we are currently unable to provide any useful information. Given the past record of Microsoft, we do not expect this information to become available.

9.7 Case Study: Multics Ring System (1965)

> Inspect now the far end, the impossible end, of the feasibility spectrum. Here we find that hecatomb called SAGE, and the other (all the other) command and control projects. We find the great corporate MIS systems – the automated board room, the self-optimizing model, the realistic management game. And, pardon my chuckles, if we turn over a few flat stones we may even find MULTICS.
>
> — H. R. J. Grosch, in DATAMATION, November 1, 1971

9.7.1 The Aged Aged Man

In 1965, six papers describing Multics were made available to the Fall Joint Computer Conference. They described a new, as yet unimplemented operating system implemented in a high level language (PL/I), which made use of virtual memory. The proposal was seen as overambitious, inspiring comments like the one in the introduction to this case study. Nevertheless, the system was developed by GE and eventually became available to the paying customer, MIT, in October 1969. In 1970, GE sold its computer business to Honeywell, which continued to sell a few dozen installations of Multics as a commercial system.

Development continued apace until the late 70s, and it was not until 1985 that Honeywell finally managed, at a sixth attempt, to cancel the product. The last remaining Multics installation was at the Canadian Department of National Defence in Halifax, Nova Scotia, Canada. It was decommissioned at 17:80Z on October 30th, 2000. It had been modified to be Y2K compliant. (Information from [TVV03]).

9.7.2 A Very Curious Thing

[Hon70] describes the three types of access control mechanism in Multics, all of which operated simultaneously, and all of which must simultaneously and individually allow access before an operation is permitted. They were "*Discretionary access control*", an ACL-like system allowing users to control access to their files and segments, "*Nondiscretionary access control*", which allows a system administrator to create restrictions covering the whole system, and "*Intraprocess access control*" or the "*Multics Ring System*", a mechanism for controlling the access of processes to the hardware and to protected subsystems. ([Hon70]). It is this last, the '*ring system*' with which we concern ourselves in this case study.

The Multics Ring System is clearly an immediate principal model, yet it also implements an unusual variant of stack inspection, as studied in section 8.6. It is notable that Multics predates by 30 years any other system using stack inspection.

9.7.3 Identification of the Current Principal

Rings are privilege levels numbered from 0 to 7. A lower numbered ring is a higher privilege level. The ring mechanism can be pictured as a series of concentric circles, with the higher privilege rings towards the centre and the lower privilege rings around them. The central supervisor routines of the system are in ring 0. System routines and nonsensitive administrative routines are in ring 1. User processes tend to use ring 4, although rings 3 through 7 are made available to them.

Any executing process consists of a number of segments. Each segment represents a procedure, or set of procedures, with defined entry points. The call chain of the process may be represented by an ordered list of segments. Each of these segments executes at a particular privilege level, that is to say, in a particular ring. Since the privileges of the system are associated with these segments, we will consider these segments to be the principals, and we will consider the immediate principal, or immediate segment, to be the current principal, thus the objects accessible to any thread of execution are those accessible from the ring of the immediate segment.

We have suggested that each segment lives and executes within a single ring. This is not quite the case. Every segment has associated with it a ring bracket, a triple (t, b, g) with $t \leq b \leq g$. The elements (t, b) specify the execution bracket of the segment, and represent the privileges of the segment. The segment may execute in any ring r such that $t \leq r \leq b$.

In order to demonstrate that Multics ring system implements a stack security model, we must demonstrate that any new segment added to the call chain cannot execute in a ring higher

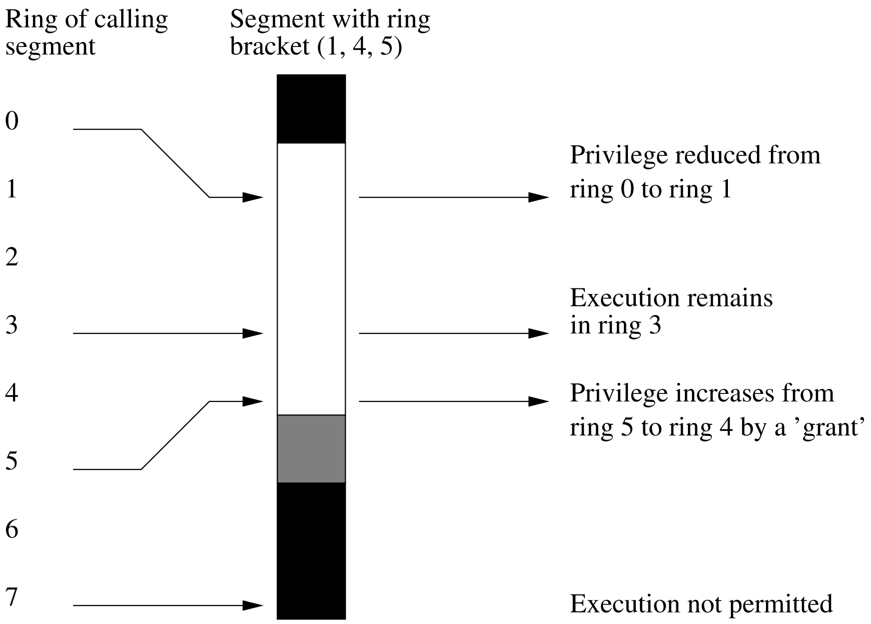

Ring of calling segment
Segment with ring bracket (1, 4, 5)
0
1
2
3
4
5
6
7
Privilege reduced from ring 0 to ring 1
Execution remains in ring 3
Privilege increases from ring 5 to ring 4 by a 'grant'
Execution not permitted

Figure 33: The four possibilities for a Multics segment call

than that of any previous principal on the call chain unless this privilege is explicitly granted to the call chain. We do this by considering the possibilities for calling a new segment individually, and there are four such possibilities.

1. If a new segment is called by a segment executing in a ring within the ring bracket of the new segment, then the new segment continues to execute in the same ring. For example, a segment with ring bracket $(1, 5, g)$, when called from a segment in ring 4, will continue to execute in ring 4, but when called from a segment in ring 1, will continue to execute in ring 1. Thus under normal operation, all segments in the call chain will execute within the same ring, satisfying our hypothesis.

2. If the calling segment is executing in a ring r such that $r < t$, that is, it is executing in a ring of higher privilege than the ring bracket of the new segment, then the new segment will execute in ring t. Privileges are lost by the call; the immediate segment has lower privileges than all preceding segments, and thus our hypothesis is satisfied.

3. If the calling segment is executing in a ring r such that $b < r \leq g$, that is, it is executing in a ring of lower privilege than the ring bracket of the new segment but of privilege no lower than that specified by the third element of the ring bracket, g, then the call chain is *granted* the privilege specified by the bottom of the ring bracket, b. Normally, a segment will have a ring bracket such that $b = g$, so that no grants are possible by calling the segment. This variant of the call models the explicit grant operation in the stack inspection calculus, and it is assumed that any programmer creating a segment with a ring bracket such that $b < g$ will verify arguments passed to the segment.

4. If the calling segment is executing in a ring r such that $g < r$, then the call is disallowed.

9.7.4 Vulnerabilities

For its age, Multics is a remarkably well designed system with considerable flexibility. However, by analysis and with the assistance of our models, we may identify the vulnerabilities present in its mechanism for the identification of the current principal.

Vulnerabilities: Vulnerability 3 in section 8.19 (the untrusted value) applies, as usual in any stack inspection system.

Batch processing was added to Multics as an afterthought. However, the security context under which a batch job was created was not maintained, as it is for callbacks under Java. The famous Multics hack involved creating a batch job for the card reader to write an executable into the path of a victim user. The victim would execute this code with their privileges, and the code would then be free to misbehave.

Vulnerabilities: If grants are used or context is discarded, the vulnerability 2 in section 8.19 (the untrusted caller) applies.

9.7.5 Modification of Rights

The current principal identified by the ring system is only used for controlling interprocess communication and access to system resources. There are no rights, and hence no rules for their modification.

The Multics discretionary access control mechanism contains an entirely separate concept of the current principal and an entirely separate set of rules for the modification of rights. However, it is a fairly convoluted access matrix using ACLs, some details of which are mentioned in section 7.12.1. The current principal is the fixed access identifier of the current process. These mechanisms are not of interest.

9.7.6 Summary of Multics

The cycle of history from Multics to Unix to Java demonstrates that the world turns full circle, even though we may not be aware of it at the time. One generation passes away, another generation comes, but stack inspection, of a form, abides forever.

9.8 Data Inspection Based Mechanisms

Of all our stack inspection systems, Java and the Java standard class library stand above the rest as an extremely versatile partnership. Any part of the mechanism for the identification of the current principal may be superseded, and any such modification would invalidate our study. However, the available mechanisms are restricted by the support provided by the virtual machine. There are certain parts of the standard Sun Microsystems JVM which may not be modified by user code. These include core functions such as the allocation of memory and the instantiation of objects. It is this limitation which prevents us from implementing even a primitive data protection model.

There do exist JVMs which allow access to VM features at this low level, the most prominent and exciting offering being the IBM Jikes RVM ([IBM03]). However, in order to get to grips with a practical implementation of data inspection, we must study Perl.

9.9 Case Study: Perl (1987)

> The manual is now not lying when it says that perl is generally faster than sed. I hope.
>
> — Larry Wall, Perl 1.000 release notes

> . . . a language in syntactically complete when you find a use for every bit of the keyboard.
>
> — attributed to Larry Wall, Usenet

9.9.1 The History of Perl

In 1987, Larry Wall released Perl 1.000 as an "*interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information.*" Increasing numbers of developers and documenters joined the project, and the language rapidly became popular. In January 1991, the first edition of the "*Camel Book*" was published by O'Reilly and Associates to accompany version 4.00 of the language. On 18th October, 1994, Perl 5 was released. It was a complete rewrite of Perl, including language support for support for object oriented programming, pointers and the many other features familiar to Perl programmers today.

In 1995, Jarkko Hietaniemi reincarnated the idea of a centralised archive for all contributed Perl scripts and modules, and CPAN was born.[73] The "Perl Bookshelf" also grew, as many of the primary Perl developers and maintainers wrote texts, and Jon Orwant started "The Perl Journal", a slightly irregular publication dedicated to Perl. The introduction of a wide range of supporting software, including mod_perl, PerlScript and CGI.pm, extended the usage of Perl into many new fields, and O'Reilly extended the Perl bookshelf to cover many of these new areas of application. (Information from [Ash02], [Hie01], [Fou04] and [Inc04])

9.9.2 Perl for System Administration

It is difficult to identify a single application niche where Perl belongs. However, one of the primary uses of Perl is still as a scripting language in Unix system administration. A Perl program is a Unix process, and is therefore a principal within the Unix model described in section 9.2. A Perl program is therefore potentially vulnerable to being exploited as a middleman by a malicious caller, with a malicious value, or by a malicious callee. This risk is especially pronounced

[73] Rasmus Lerdorf also wrote a Perl CGI script which collected information on visitors to his home page and called it "PHP Tools", for "Personal Home Page". PHP was born as a bastard child of Perl.

when a Unix process is performing system administration tasks, since such processes usually run with *root* privileges, and thus have the power to modify any part of the system.

Perl allows the programmer to considerably reduce the risk of such exploitation by including a '*data tainting*' security model, the primary requirement of which is that "*you may not use data derived from outside your program to affect something else outside your program–at least, not by accident.*" ([Per01]). When this security mechanism is in operation, it is possible to read data from untrusted sources, be that source a caller or a callee, but not for this untrusted data to influence anything outside the process itself. Thus it is impossible for a Perl process protected by this mechanism to be exploited as a naïve middleman in a Unix call chain. Data tainting was initially introduced in section 8.7.2

9.9.3 Identification of the Current Principal

The Perl tainting mechanism is an implementation of a data inspection calculus with only one privilege, '*untainted*', represented by the absence of a taint bit associated with any particular value in the mutator. We might equivalently consider this system to have two principals, the `tainted` principal with no privileges, and the `untainted` principal having the one and only privilege.

Data provided by any source external to the process is marked as *tainted*. In our data inspection calculus, this would be framed by the `tainted` principal. This externally sourced data includes but is not limited to data passed to the process on the command line, read from environment variables, the results of many system calls, values received by IPC, and anything read from a file. Any data derived in computation from tainted data or a mixture of tainted and untainted data is considered to be tainted. Only data provided entirely by the process itself is considered to be untainted.

Many operations may not be performed with tainted data. These include, with one or two documented exceptions, all commands that invoke subshells or modify files, directories or processes.

It is possible to untaint data by matching it against a regular expression and extracting a subpattern. [Per01] says about this, "*The only way to bypass the tainting mechanism is by referencing subpatterns from a regular expression match. Perl presumes that if you reference a substring using $1, $2, etc., that you knew what you were doing when you wrote the pattern.*"

9.9.4 Specifics of Implementation

The Perl documentation is of such a high quality that it seems sufficient just to quote the specifics of implementation from it. Much of what remains in this is extracted almost verbatim from [Per01].

Perl automatically enables taint mode when it detects its program running with differing real and effective user or group IDs. The taint mode can also be enabled explicitly by using the -T command line flag. Once taint mode is enabled, it cannot be disabled. When taint mode is enabled, Perl performs a number of security checks, ranging from simple precautions, such as verifying that path directories are not writable by others, up to and including the full data security model described above.

Tainted data anywhere within an expression renders the entire expression tainted. It would be inefficient for every operator to test every argument for taintedness. Instead, the slightly more efficient and conservative approach is used that if any tainted value has been accessed within the same expression, the whole expression is considered tainted ([Per01]).

9.9.5 Summary of Perl

Perl implements an impressively practical data inspection mechanism. It is sufficiently fast and efficient that [Per01] states, "*This flag is strongly suggested for server programs and any program run on behalf of someone else, such as a CGI script.*" It is slightly conservative, since the finest granularity available to the tainting mechanism is the expression; however this granularity is traded off against increased speed at runtime. The untainting mechanism also encourages the programmer to consider precisely what is being untainted, rather than simply untainting an entire value.

For all this, the Perl data inspection model has only one clearly defined purpose, and that is to patch some of the vulnerabilities discovered in the Unix model of section 9.2. It is not a general purpose solution for the management of principals and privileges since it has only two principals.

The Perl model achieves its objective, but the objective is limited beyond what we would ideally desire.

9.10 More than Tainting

Data inspection models are not widely used because of the considerable runtime overhead that even the simpler data tainting models can impose. Perl, possibly the most widely data inspection system, does not universally impose the tainting mechanism: it is enabled only when explicitly requested by the user, or when the Perl interpreter runs with greater privileges than the Unix process which invoked it. A data tainting model was also used in JavaScript version 1.1, providing a possible alternative case study. However, as further evidence of the lack of popularity of data inspection models, David Flanagan documents in [Fla98]:

> Data tainting was a security model available on an experimental basis in Navigator 3. The experiment proved unsuccessful and it was removed in Navigator 4.

Tainting is a very limited and restricted implementation of data inspection security. They manipulate only one right and two principals, and this in itself makes them a poor example for studying models for the dynamics of rights, as described in section 7. Data inspection models have been implemented on a grander scale and in such a way that a model for the manipulation of rights may become apparent. '*Capability*' systems attach capabilities, that is, privileges, to many objects, and the rules for the manipulation of capabilities satisfy the requirements for a data inspection system.

9.11 Case Study: EROS and Capabilities

> [We define] the term '*principal*' to mean an individual [...] to whom charges are made for the expenditure of system resources.
>
> — J. B. Dennis, in [DvH66], getting the right idea!

9.11.1 History of Capabilities

Capabilities appeared surprisingly early in the development of protection systems, well before Multics. In 1966, [DvH66] produced an extensive formal definition of capabilities and programming on a capability system designed for resource control in a time-sharing system. Capabilities were subsequently implemented in many systems, including Hydra/C.mmp: a descendant of Hydra ([WLH81]), CAL ([LS76]), Plessey 2000 ([Lev84]), Sigma 7, the B5700 ([Org73]), CAP ([NW74], [NH82]), KeyKOS ([Har85], [RHFL86]) and EROS ([Sha99], [SSF99]).

The full capability model, while correct, suffered from severe difficulties in implementation ([CGJ88], [FL94]), including problems related to memory management, computational overhead and the lack of hardware support. None of these issues is obviously insurmountable, and some are now considered solved. Nevertheless, the only capability system to become a major commercial success was the IBM AS/400 ([Sol96]), which had appropriate hardware support in both the instruction set and the memory management system. (Information from [Sha99] and [SSF99]).

Capability systems specify a data-oriented mechanism for the manipulation of rights in the protection system. The mechanism for the identification of the current principal implemented by a capability system is very similar to that formalised by the data inspection calculus.

9.11.2 Identification of the Current Principal

A '*capability*' is defined in [DvH66] to be an unforgeable pair (`object`, `right`); it is no coincidence that this matches our definition of a permission in definition 2.9. Every process has available to it a list of capabilities. In order to access any object in the system, a process must hold and invoke a capability for that object ([Sha99]). The simplicity of the interface to the kernel means that a process has no way of requesting any operation for which it does not hold a capability; the only interface available is '*invoke capability*'. The current principal of a capability system is therefore the immediate principal, since only the capabilities of the immediate principal feature in access control decisions.

The use of the immediate principal as the current principal does not divorce capability models from the data inspection model described in section 8.12: this immediate principal is not atomic! In fact, even in the data inspection calculus, a guarded substitution considers only the frames around the immediately presented value when making a protection decision. In order to understand this model, we must therefore understand how capabilities are generated and how they may be acquired by processes.

Just as certain system calls under Unix return descriptors, so do some capabilities, when invoked, generate further capabilities. However, these new capabilities will not have any privileges which the original capability did not permit. For example, a read-only directory capability can generate only read-only file capabilities, and as a slight special case, a '*weak read*' capability ([SSF99]) for a table of capabilities will demote all capabilities read from the table to read-only and weak, thus limiting access propagation of write capabilities over transitive read capabilities. Thus the capability mechanism is in agreement with both the the Principle of Attenuation of Privileges from section 2.4.2 and the data inspection calculus of section 8.12. Any new capability may have at most the privileges inherited from the capability from which it was generated.

9.11.3 Specifics of Implementation

The rules for the derivation of new capabilities implement a data inspection model. However, this implementation may be vulnerable if the capability model is not ubiquitous. If the system contains any unframed data, it may be possible for a process to acquire a filename from a low privilege file descriptor and use this to generate a high privilege file descriptor for the same file. A malicious process supplying such an unframed filename may be able to generate and use a high privilege descriptor within a trusted process, thus exploiting the vulnerability.

Vulnerabilities: If unframed data is permitted in the system, either of vulnerability 2 in section 8.19 (the untrusted caller) or vulnerability 3 in section 8.19 (the untrusted value) may be introduced.

Capabilities may be transmitted from process to process, thus effecting a grant operation. However, revocation is difficult and EROS provides only a primitive revocation mechanism. Every object and every capability has a version number. If the version number of a capability does not match that of the object to which it grants access, then it is considered void. By incrementing the version number on an object, all capabilities to it may be revoked ([SSF99]). The documentation suggests using an intermediary process in any situation where selective revocation is required. However, this intermediary process must solve the entire access control problem anew; the capability structure provided by the operating system is likely to be of little assistance in this case.

9.11.4 Summary of Capabilities

The capability model is an elegant model, and within the world of descriptors and capabilities, it is a correct implementation of the data inspection calculus described in section 8.12. Performance measurements in [SSF99] indicate that capabilities are a sufficiently fast mechanism to be used in practice. However, with the notable exception of AS/400, they have yet to become successful in any major implementation.

9.12 On the Granting of Rights

It is important for an implementor to remember that the two problems of identification of the current principal and access control are separable, and may be implemented separately. Yet so far, EROS is the only system which permits any flexibility in the reassignment of privileges, and it permits this largely as a natural consequence of the mechanism for the identification of the current principal and standard calling semantics. We have not yet seen any effective, dedicated and dynamic privilege management system.

We will describe an implementation of a stand-alone access control and privilege management system which implements an '*ideal model*' very similar to that described in section 7.10. Even with completely independent implementations of the two stages of access control, it is still possible to design an interface between this privilege management system and the mechanism for the identification of the current principal such that a tristate logic may be implemented, as described in section 8.10. The current implementation of this system uses a stack inspection mechanism for the identification of the current principal, but it could easily use a data inspection mechanism or even a trivial immediate principal mechanism. This case study will demonstrate that an elegant system for the management of rights may be implemented even *without* an implementation of one of the more complex schemes for the identification of the current principal.

9.13 Anarres II

```
jess:  /* Watch carefully */
jess sniggers
```

— jess, finding comments in the source code of Anarres II

9.13.1 History of Anarres II

Anarres II ([Man03], [Man97]) started as a small project in about 1997, and rapidly grew into a fully fledged Unix-like system written in LPC for the MudOS virtual machine. It uses a flash compiler and bytecode interpreter which provides some basic reflection. All code in the system may be modified at runtime without restarting the system, with the exception of the reference monitor built into the interpreter.

The Unix user and group systems initially used for the representation of principals were inadequate to the task of managing permissions for a dynamic system, and in about 1999 they were replaced with a transitive RBAC model allowing full delegation of authority to the users. Since that time, the superuser, '`root`' has needed to log in on only a handful of occasions to perform reboots or major system updates.

The system has evolved beyond its experimental roots, and work is underway to write a new compiler and runtime library to increase the scalability of the system to handle thousands of simultaneous users.

9.13.2 Modification of Rights

Anarres II is a level 3 transitive system (see section 7.11) which uses stack inspection to identify the current principal. However, we have studied stack inspection in depth. Anarres II also has a fully developed mechanism for the modification of privileges which is based on the '*ideal*' model of section 7.10. The object responsible for managing privileges within the system is called the '*privilege daemon*'.

The functional interface to the privilege daemon includes only the functions `create principal`, `destroy principal`, `grant principal`, `revoke principal` and `compare principals`, the minimal possible interface such that a current principal may be computed and an access decision made. These operations are the fundamental operations on privileges, and no assumptions may be made about the implementation. Specifically, it is not required that the privilege daemon evaluate full principals to perform the comparison.

Anarres II uses a transitive access right r; a full principal is discovered by performing a graph walk over the right r from any principal. Every principal has a unique '*grant role*', as described in theorem 7.2 in section 7.10. The right r to any principal p may be granted if the user has access to the grant role x for that principal: this grant role is the principal x such that $g \in [p, x]$, just as in the level 2 transitive model. Anarres II also has a third right, called c, for '*control*'. The control right identifies a unique '*control role*' for each role, and it is this extra c right which makes Anarres II a level 3 transitive system. Access to both the grant and control roles confers the ability to change the grant role for any given principal, and access to the control role alone confers the ability to change the control role. This is justified below.

Principals in Anarres II form an artificial tree; thus every principal also has a parent, which implicitly has access via r to all its children. The top of the tree has the special name `root`, thus `root` automatically holds all privileges even though none have been explicitly granted. There are currently four sub-trees under `root`; these are called `domain`, `group`, `sys` and `user`. Within these trees, a big-endian hierarchical naming strategy similar to that of the DNS ([NWG82]) is used, so the principal `sys.file.doc.area` is an immediate subprincipal of `sys.file.doc`. Privileges within the `domain` hierarchy tend to be used for access to elements of the game world. The `group` hierarchy is used for generic groupings of users and projects, and is generally a leftover from the older Unix-like '`/etc/group`' system. `sys` privileges give access to operating system features or parts of the filesystem.

The `user` hierarchy has in turn a sub-hierarchy for every registered user. The principal `user.arren` is a user role, as described in theorem 7.1 on page 120. Any privileges granted to a user will be granted to that user role. In the current configuration, a user has control of their privilege, but may not grant it. Thus any user is free to extend the hierarchy rooted at his user role, and grant any subprivileges as he pleases, but may not grant his user role. This allows users to enforce arbitrary mandatory access control on any objects they create, while enforcing some amount of sanity: any user having a particular user role *must be* that user. The user may not change the grant role of his user role, since that requires access to the existing grant role.

The safety problem for Anarres II may be decided by a graph walk over the r and g privileges;

this is almost identical to algorithm 2 on page 129. In fact, the algorithm for the evaluation of the current principal in Anarres II is that from algorithm 4 on page 4, where a comparison operator on principals may be used instead of the condition $|P - principal(f)| > 0$ from that algorithm.

9.13.3 Specifics of Implementation

Anarres II implements a semantically preserving cache of principals inside the privilege daemon, allowing a full principal to be evaluated in an average of 3000 instructions; this average decreases as system uptime increases due to the '*hot*' node caching strategy described below. This cache is made semantically preserving by selectively flushing parts of the cache whenever a modification to a principal is made. The cache itself stores enough information to decide which parts should be flushed. This operation is not common, and may be performed fast.

The privilege daemon performs a full graph walk internally in order to decide whether a given principal has a given privilege. Some thought was given to aborting this graph walk as soon as the privilege was discovered, but there is more benefit to be obtained from caching the full principal than from attempting to represent a partial principal such that the graph walk may later be continued.

Caching of principals can be used to speed up the evaluation of further principals. The cache keeps a record of which principals are '*hot*': principals which are accessible to many other principals, and thus frequently traversed when evaluating a full principal. Explicit caches for these principals are built incrementally, even if they are never explicitly requested. Any principal having access to a hot principal may now simply copy its access cache instead of re-performing the entire graph walk from that principal. This particular optimisation has a significant effect on the performance of the system since most users directly hold only a few project privileges, and these tend to be the hot principals.

The Anarres II privilege system does not represent a full lattice; it only represents the top half of the hierarchy. This means that explicit privileges must be created whenever a 'bottom' privilege is required. This will be remedied in a future release. The rights g and c are also not explicitly represented as such. Since, for any principal p, there is exactly one principal x such that $g \in [p, x]$, the name of x may be stored in the structure representing p. The same applies to c.

For reasons of sanity and administrative convenience, the privilege daemon restricts grant and control roles thus: if A is a child privilege of B then $g(A)$ must be a subprivilege of $g(B)$. The restriction for control roles is identical.

9.13.4 Summary of Anarres II

Anarres II is a highly practical and scalable system for managing principals and privileges. The implementation provides interfaces only the minimal interface for the creation, destruction and test for existence of privileges, yet this is sufficient for all practical purposes. Furthermore, this separation of concern allows the mechanism for the identification of the current principal to be replaced without modification to the privilege daemon.

The aim of the Anarres II protection subsystem is to provide a correct and practical protection system for an operating system which is flexible to the changing needs of the users and does not require the regular intervention of a superuser. It succeeds in this objective.

9.14 Summary of Case Studies

Many of the systems described in this section have good approaches to security, some of them surprisingly good. In summary, we saw:

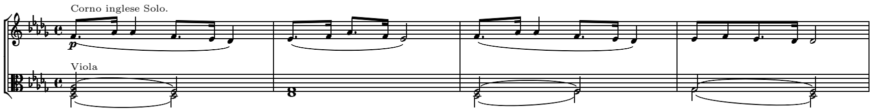
- **Unix:** An opaque UID based system with paranoid, yet undecidable, dynamics of rights. While this appears the simplest to implement, this is not actually the case.
- **Java:** A stack inspection mechanism under which rights may not be granted. The Java security model is still developing, and much may be learned from it.
- **Multics:** A stack inspection mechanism 30 years before its time!
- **Perl:** A data inspection mechanism with only one right. Perl makes data inspection practical, a rare feat.
- **EROS:** A data inspection system which appears surprisingly elegant but has yet to become successful.
- **Anarres II:** An example of effective privilege management which shows that the utopia of expressiveness and distribution may be achieved.

It appears that nobody has yet combined all of the desirable elements from these systems into a single complete and correct protection mechanism. Many of the problems which plagued systems in early years have now been solved; memory management, cryptography, virtual memory – there are now off-the-shelf solutions to all of these problems.

Material which follows from this section may be found in:

- Section 10: Having seen where others have gone wrong, where they have strengths and where they have weaknesses, we are in a far stronger position to apply our theoretical knowledge to the problem of *designing* a new model for security.
- Section 11: We have some ideas as to *why* no existing security model provides the utopia of correct and practical security, but we save this for the closing address of our conclusions.
- Section 12: We have some more ideas for fast and correct implementations of data inspection mechanisms, which we leave for further work.

So what do we do with all these shiny toys . . . ?



10 A New Model for Computation in Protection Systems

There are two possible applications for a fuller theoretical understanding of protection. The first, which we have seen already, is an analysis of existing protection systems to identify flaws and weaknesses. The second, only partially explored possibility is that we might be able to design a protection system from the ground up to satisfy a given set of desirable requirements.

We started in section 7 to design a practical '*ideal*' protection system, but we could not complete this design without sufficient understanding from section 8 of the concept of the principal and the mechanism for the identification of the current principal. Given this understanding, we may now unify the structures and concepts from these two sections and construct a new computational model for the principal. This model permits us to construct a single protection system which simultaneously encapsulates the all the best properties of our previous models both for the identification of the current principal and for the management of privileges.

The system we propose is surprisingly simple but provides a demonstrably powerful security model with the ability to automatically maintain protection domains, allow the transmission of rights, and hence delegation of responsibility, and provide accountability and auditing. The system may be almost fully distributed, and since it is derived directly from our theoretical models, we will immediately have the ability to prove conformance to a security policy in polynomial time.

We will describe in section 10.1 the foundation of this design: a new representation of the principal which has many convenient computational properties. We will outline some theoretical details behind an implementation in section 10.2, and we will consider some explanatory scenarios and case studies in section 10.3.

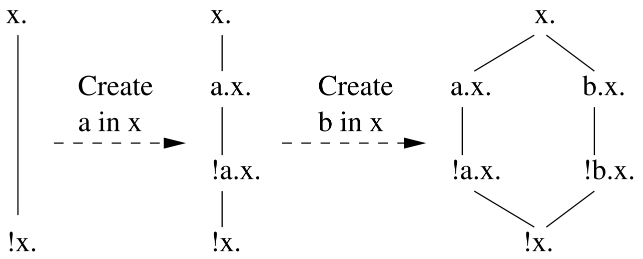


Figure 34: The Creation of a New Principal

10.1 The Lattice of Principals

Our proposed design is centred around a representation of the principal which unifies the concept from the λ_{sec}-calculus of the principal as a set of privileges, with the concept of the principal as an opaque identifier for a graph node, as in the ideal model. There is a surprising duality between the data inspection λ_{sec}-calculus for the identification of the current principal, as described in section 8.12, and the 'ideal' mechanism for the manipulation of rights, as described in section 7.10. While these models serve orthogonal purposes, it is possible to unify representation of principals between the two mechanisms in such a way that the operations required both models are computable in $O(1)$ or $O(n)$ time. This unified representation is called the "*Lattice of Principals*". This lattice is by no means a new concept; it was encountered first as the '*diamond pattern*' in section 7.9.8 where we applied the ideal model to our normal concept of an organisational hierarchy.

10.1.1 Background to the Lattice of Principals

The configuration of a transitive '*ideal*' protection system, as described in section 7, has the properties that everything is a principal, and the access relation is transitive. The operations on the configuration are creation and destruction of principals, granting and revocation of rights, and evaluation of access within (or safety of) a configuration.

The λ_{sec}-calculus computes with sets of privileges, which are, by definition, principals. The only operations on these sets are the subset operator for evaluating access (which happens to be a transitive operator), the union operator for augmenting a principal with another principal in the *grant* or *untaint* operations, and the intersection operator used in the computation of the current principal amongst multiple responsible parties.

We explicitly unify these two models by constructing an artificial lattice of principals. This lattice is given semantics such that we may express both the λ_{sec}-calculus and the ideal model simultaneously within it.

10.1.2 Operations on the Lattice of Principals

The principals of the λ_{sec}-calculus are nodes of the subset lattice. The principals of the transitive model have no explicit structure, but sufficiently good meet and join relations may easily be imposed upon the directed graph formed by the relation r that the structure of the configuration matches the subset lattice of the λ_{sec}-calculus. It is then possible to implement both the λ_{sec}-calculus of section 8.12 and the ideal model of section 7.10 over this lattice of principals.

The operations required by the λ_{sec}-calculus are subset, union and intersection on principals. The operations required by the ideal model are the creation and destruction of principals, the granting and revocation of rights and the evaluation of the transitive access relation. However, the subset operation of the λ_{sec}-calculus is identical to the access relation of the ideal model, thus there are five operations in total, of which this comparison is the most complex. We will discuss these operations individually.

The comparison of principals, equivalent to the subset operator of the λ_{sec}-calculus or the access relation of the ideal model, is a test that a given principal, usually the current principal, has a particular privilege. Conveniently, the subset comparison operator on the privilege-set principals of the λ_{sec}-calculus is semantically identical to a comparison using the transitive relation expressed by r on the configuration of an ideal system. In fact, if a principal in the configuration of an ideal system is defined by the set of principals to which it has access via the transitive right r, the two operators are precisely equivalent. This comparison may be computed in the lattice by performing a graph walk over direct access relations, following both implicit and explicit relations. This graph walk is very similar to that described in algorithm 2 on page 129 and may be performed at worst in linear time.

Creation and destruction of principals may be performed by creating or collapsing a sub-lattice at any point in the hierarchy, as in figure 34. In the first case, the new principal is the only child of the parent node `x..`. In the second case, there are existing children and the new principal is created alongside them. The removal of a principal is the inverse of this operation. These operations may both be performed in constant time.

Granting and revocation of permissions may be performed by representing the existence of the permission as an explicit relation on the lattice. This operation may be performed in constant time.

Intersection is required by the *pure* λ_{sec}-calculus both for the computation of the current principal and for restricting grants. However, the lazy evaluation strategy for the identification of the current principal in section 8.17.3 removes the requirement for intersection, and the restriction on granted principals will be provided by this algorithm. Thus, in any practical implementation, intersection of principals is not required.

If absolutely necessary, a conservative implementation of intersection may compute the greatest principal such that both of the given principals have access to it, but *usually* the bottom principal of the smallest sub-lattice containing both principals will suffice. This bottom principal may be computed in constant time. As a special case, if either principal is greater than the other, then the intersection may optionally be taken as the lesser of the two.

The union of two incomparable principals may not be computed in the lattice since any principal representing the union would also have access to itself; the result follows by contradiction. However, we can construct an alternative operation which is adequate for the purposes of the λ_{sec}-calculus. The union of principals is used by the *untaint* or *grant* operators when S grants R to a current principal P to give $(P \cup R) \cap S$. S is the immediate principal of the grant, and the granted principal R is a subset of S.[74] The current principal after the grant is therefore at most S, and at least R. From experience, it usually suffices to assume that $((P \cup R) \cap S) = R$. In the rare cases where some privilege $p \in P - R$ is still required, it is usually possible to identify or create a principal $R' \supseteq R \cup \{p\}$ which may be granted instead of R. We may therefore implement this mechanism of the λ_{sec}-calculus in constant time.

Thus we have shown that all the operations of both the ideal model of section 7 and the λ_{sec}-calculus of section 8 may be implemented over an artificial lattice of principals. The linear time algorithm for the comparison of principals is the slowest of the five operations, but this performance estimate may be improved considerably in many cases. The algorithm for performing this walk has been implemented in Anarres II, and some notes on the optimisation of this algorithm in practical implementations may be found in section 9.13.

10.1.3 An Example Lattice of Principals

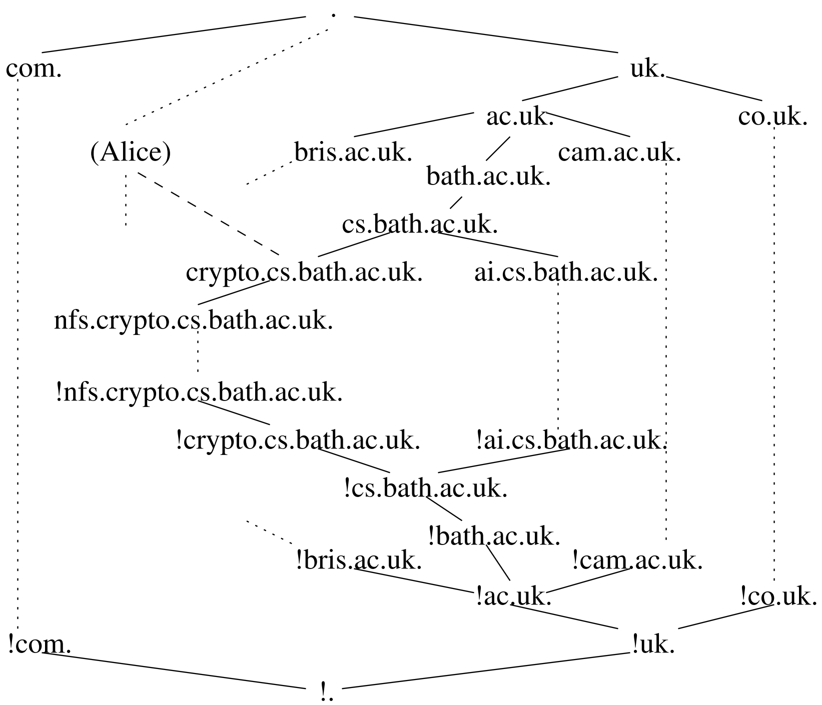


Figure 35: A Lattice of Principals

A fragmentary example of an artificial lattice of principals is illustrated in figure 35. The top half of this lattice is based on a DNS-like hierarchy ([NWG82]), which is mirrored underneath to create the bottom half of the lattice. Every node in the top hierarchy has a corresponding bottom node. At the very top of the lattice is the superuser: a principal called '`.`'. Immediately below `.` are principals representing major organisations and authorities. Each of these principals may have sub-principals, and each sub-principal may have lesser sub-principals, and so ad

[74] By the distributive law, $(P \cup R) \cap S = (P \cap S) \cup (R \cap S)$; without loss of generality, we may assume that $R \subseteq S$ since these are both full principals and are known at compile time.

infinitum.[75] In figure 35, the principal `cs.bath.ac.uk.` has sub-principals `ai.cs.bath.ac.uk.` and `crypto.cs.bath.ac.uk.`. Any principal implicitly has access to all sub-principals of itself by the transitive relation r; thus the root principal at the top of the lattice may access any principal in the lattice, and the principal `cs.bath.ac.uk.` may implicitly access anything within the sub-lattice rooted at that point.

The lower half of the lattice contains principals with names prefixed with '`!`'. These are the '*bottom*' principals described first in the '*diamond pattern*' of section 7.9.8, and their purpose is described further in section 10.2.2 below. The implicit access to sub-principals extends over bottom principals such that every '*top*' principal has access to the corresponding bottom principal and any bottom principal has access to all bottom principals below it in the inverse hierarchy.

Privileges may be granted explicitly using the rights r, g, and possibly c, as in section 7.10. We have exemplified this in our diagram by including a user 'Alice',[76] to whom the privilege `crypto.cs.bath.ac.uk.` has been granted, giving her access to everything accessible to that principal. This makes Alice a member of the cryptography group and gives her access to all resources within it. This mechanism is identical to that for granting rights in the ideal protection system of section 7.10.

A partial practical implementation of this combined model is described in the case study of Anarres II in section 9.13, but we describe the structure here in a little more detail, since it is core to our example.

10.2 Consequences of the Design

10.2.1 Representation of Principals and Privileges

Any principal may be represented unambiguously by its name in the hierarchy. While some central agreement or authority is clearly required for the allocation of names in the lattice, this authority has no further part to play in the operation of the protection system itself, and thus should not represent a failure to meet the requirement of distribution.

As the host protecting an object is the only trustable reference monitor for that object, so any principal is the only trustable authority for which other principals are explicitly granted access to that one principal. Thus the explicit relationships between principals may only be published by the principal being accessed (or any authority responsible for that principal). A principal may claim to have certain privileges $r \in [s, o]$, but these must be verified in computation by checking with the target principal o.

Since computations of privileges are only performed by any system only for the purpose of accessing objects protected by that host's reference monitor, any miscalculation will immediately affect only the security of that one host, and subsequently any hosts which trust that one

[75] With apologies to Augustus de Morgan and his fleas.

[76] The brackets around the name 'Alice' in the figure indicate that the name 'Alice' is not fully qualified in the hierarchy. It is likely that user roles such as 'Alice' will be placed almost immediately within the organisation which authenticates them. Thus Alice might be `alice.user.bath.ac.uk.` or even `alice.user.uk.` if the government gets its way.

host. It is therefore in the interest of all hosts to make sure that all computation with principals is performed correctly and that any privilege caching strategy preserves the semantics of the underlying implementation.

10.2.2 Sharing Data in the Lattice of Principals

The concept of a hierarchy of access is familiar to us. Less familiar are the structures required to facilitate sharing. Usually, we share data by creating a principal with which to protect the data and granting access to every principal with whom we wish to share the data. However, when the number of principals within a sub-organisation is both rapidly changing and possibly uncountably infinite, we must have a new strategy, and this strategy is the justification for the bottom nodes.

Every principal of the upper hierarchy is mirrored to produce a corresponding '*bottom*' principal in the lower half of the lattice, the name of which is prefixed with a `!`. The *bottom* principal `!x` is the principal which is accessible to all principals in the sub-lattice rooted at `x`. Thus any administrative unit will be represented by a sub-lattice, and will have a top node, which controls the unit, and a bottom node, which may be accessed by all members of the unit. This pattern repeats for sub-units and sub-sub-units within those, again ad infinitum.

It is now possible to grant a privilege to all the (possibly uncountable) principals in any sub-lattice by granting the privilege to the bottom principal of that lattice. Since all principals in the sub-lattice have access to the bottom principal of the lattice, they will have the new privilege by the transitivity of the access relation in the ideal model.

This concept of the '*bottom principal*' appeared in the 'diamond pattern' in section 7.9.8. It is mandated neither by the λ_{sec}-calculus nor by the ideal model, but arises as a result of the requirement that we be able to manipulate uncountably infinite numbers of principals.

10.2.3 Encapsulation of Data

Any practical implementation of the data inspection λ_{sec}-calculus from section 8.12 must maintain a *frame* as metadata on every piece of code or data in a system or on the network. Such a frame indicates the responsibility of a set of principals for the framed code or data, and would be manipulated by the mutator as described in our λ_{sec}-calculus.

The major failure of all existing models is that the metadata is too readily discarded. Currently, when data is written to a file and reloaded, or transmitted over a network, frame information is lost, and any later permissions check will not be able to take into account the principals previously responsible for this data.

It is likely that current systems discard metadata because the underlying systems or protocols cannot support the richer semantics required. There are issues of compatibility with legacy applications, and there is scope for '*cheating*' or lying, especially in the case of transmissions over a network or between administrative units.

It would be relatively simple to augment the transport layer of the network to include out-of-band information about the principals responsible for the data being transmitted over a link.

Similarly, it is possible to augment filesystem metadata to include this information.

Legacy systems which do not support metadata at all can always be supported using appropriate '*black box*' interfaces, but it remains to show that metadata cannot usefully be falsified in a distributed computation.

10.2.4 Integrity of Metadata

There are only three possible '*cheats*' which might modify frames as they are transmitted over a network. However, if we assume the existence of a cryptographic signature operation, two of them are shown to be impossible, and the third is required for the normal operation of the system:

1. **A principal does not frame data which it has modified.** Any principal receiving this data knows the identify of the sender, and will notice the absence of the sender's frame.

2. **A principal removes a third principal's frame from data.** This is allowed, and is the '*untaint*' operation. If the principal does this 'wrong', then other principals' policies will eventually dictate that the principal is no longer trusted. This strategy represents a long term loss for the first principal, while never allowing it to raise its own privilege. However, this also allows for privacy to be created within a system which could otherwise expose too much metadata about a computation, as described in section 10.2.5.

3. **A principal adds a third principal's frame when that principal is not responsible for any part of the computation.** This is impossible because frames will be cryptographically protected such that they may only be generated by the responsible principal.

Therefore the metadata is effectively inviolate even in the presence of malicious hosts or storage mechanisms.

10.2.5 Protecting Internal Structure

While, within any organisation, our model can provide an extremely accurate model of principals, an organisation will not necessarily wish to expose such internal structure to any principal outside itself. The privilege information of a piece of data may expose the path a piece of data has taken inside the organisation, but a user reading a web site will not wish to know the route the page has taken through the database servers, middleware and front end servers, nor will the organisation wish to publish this information.

It is not usually important which segment of an organisation produced a message. It is only significant that it was produced by a particular organisation or sub-organisation. It will suffice for the border gateway of the organisation to reframe outgoing data with the bottom privilege of the organisational sub-lattice, since no higher privilege will ever be required of the data than to show that it originated within that organisation.

This pattern may be repeated throughout the lattice at every level; a department exporting data to the company need not say which employee created the data. The data as exported is the responsibility of the department, not an individual employee and the department bottom privilege will again suffice.

This not only avoids the exposure of unnecessary internal structure, but it reduces the amount of data sent over a network and also allows an organisation to use an alternative security mechanism internally while remaining compatible with a larger network using a lattice-based model.

10.3 Case Studies for Implementation

The concepts of principal, frame, function and value in the λ_{sec}-calculus of section 8 are entirely agnostic as to the granularity in practice of the objects they represent. A function may be a *sine* computation. It may be the transfer of a value from one register to another. Or it may be an entire process with associated inputs and outputs.

It is likely that the granularity at which the λ_{sec}-calculus is applied will be close to the granularity at which principals take responsibility for data or code: if the λ_{sec}-calculus is applied at much finer granularity, overhead increases dramatically. If much lower, then incorrect access decisions may be made.

We consider several possible granularities and cases for the implementation of this system, and comment on the advantages and drawbacks of each.

10.3.1 Virtual Machine

Virtualisation has already reached the consumer with the advent of platform independent virtual machines, such as Java ([Sun03d], [IBM03], [IG03], [F$^+$03]), Perl ([Fou04], [Inc04], [Han04]) and .net ([Mic03b]). It is becoming increasingly common for entire applications to exist fully within such virtual machines.

A virtual machine isolates the application from the underlying system, and is thus ideally placed to perform reference monitoring on all host and system resources. This layer of isolation also permits the virtual machine to implement a trusted computing base inaccessible to the application mutator, and it is common for an access control mechanism to be implemented in this trusted computing base.

It is possible to implement the protection mechanisms described in this thesis at virtual machine level. This is one of the simplest possible scenarios, since the application code need not be aware of the underlying protection mechanism. The virtual machine is also free to encapsulate all incoming and outgoing network and filesystem data in frames, thus greatly simplifying the process of migration to the lattice model without requiring modification of application code.

10.3.2 Operating System

A fairly coarse grained mechanism might be implemented in the operating system for a host. The operating system has the power to isolate processes within host virtual machines, and can manage privileges for each virtual machine within the operating system kernel. The kernel can monitor all input and output from the process, thus it can frame incoming and outgoing data, and will be aware of which principals might have influenced the computation within the process.

We do not prefer this mechanism: it is not sufficiently fine grained to distinguish between the parts of a process which may have been influenced by different sets of principals. It would be quite suitable for a system where all applications used a micro-kernel architecture, but not for current monolithic applications. This implementation does maintain the advantage that it may be implemented without modification of existing application code.

10.3.3 Application Library

It is also possible to create an implementation as an add-on library for any given target language. Rather than relying on the underlying implementation to override input and output primitives and handle framing of data, the application must explicitly call code within the library to perform these tasks. It must then explicitly combine the frames during any computation, and pass these frames as a part of any request for a protected operation.

If the application calls any primitive input functions which do not account for frames on the data, it may obtain from an untrusted source data which causes a failure of the access control mechanism. Primitive output functions which do not frame data would not pose a security risk, but any recipient of unframed data must assume that it is untrustworthy.

This implementation gives the application programmer a very high level of control over the operation of the protection mechanism. He may control the granularity as appropriate during a computation, designating entire blocks of data as trusted or untrusted, rather than performing operations on principals as overhead in every single operation. However, this implementation also carries the highest risk of programmer error and the highest overheads for maintenance.

This implementation would have the lowest initial cost, and would produce a nice proof of concept for a particular application field, while providing significant benefit for the network as a whole.

10.3.4 Scope of Implementation

Perhaps the easiest world to visualise is that where all information and code in the system is appropriately framed and every mutator obeys the conventions set out by the λ_{sec}-calculus for manipulation of these frames. This possible world is as unlikely as it would be ideal. In practice, the scope of implementation is likely to be dictated by the presence of legacy code which cannot be updated. Compatibility is certainly achievable, although legacy applications will not necessarily benefit from the newer access control mechanisms.

There are many possible scopes for implementation which work around various issues with legacy code. None of these suggestions is perfect, and each has weaknesses relating to the introduction of unframed data into the λ_{sec}-calculus.

A partial implementation might consist of a set of hosts which obey the framing conventions for all data passed between them, but permit unframed data to enter the network with a frame of `!..`. Such hosts need not be geographically or logically connected, and might communicate over the normal internet using appropriate protocols. The only weakness of this design is the possibility that unframed data might influence computation within any λ_{sec}-calculus mutator.

An organisation need not implement a complete protection framework internally. It suffices that the system provide a correct interface to the rest of the world. A system which does not wish to implement the λ_{sec}-calculus internally, either for reasons of legacy code or simplicity, but wishes to participate in a network which uses framed communication, may use a border implementation. An intelligent border router might be used to frame all outgoing data. On the assumption that all data was generated purely internally, the safe choice of frame is `!x.` where the system represents the sub-lattice `x..`. If data was generated in response to an external request, it may be possible to identify the data with the request on the system border, and attach frames from the request to the response.

An even more localised option would be the implementation of a protection framework within a single application, if the application has sufficient complexity to warrant this. Examples of such applications include collaborative or personalisable web sites, large multi-player games, or even, when considered as an application in its own right, a virtual machine interpreter like Java. As noted in section 10.3.3, there is a much wider scope for programmer error without a trusted computing base. However, the application of a good security model even can greatly simplify privilege management and give well defined security properties within a complex application.

10.4 Summary

We have shown how elements of the theory of protection systems may be combined to build a practical and applicable protection system satisfying many requirements not satisfied by protection systems in use today. The system builds upon elements of the λ_{sec}-calculus from section 8 and the ideal models of section 7, and combines these using a representation of the principal which is richer than that specified by either model. This representation of principals has the following properties:

- **Principals need not be countable**: There is never a need to enumerate principals.
- **Privileges need not be countable**: A privilege is represented by pair of named principals and a right, and these need never be enumerated.
- **Principals are easily serialisable as metadata**: A principal is a name by which associated metadata may be referenced.
- **Distribution is easy**: Portions of the lattice may be delegated and re-delegated like DNS.

- **The system is computationally simple**: All the operations required by both the ideal model and the λ_{sec}-calculus may be implemented in at worst linear time.

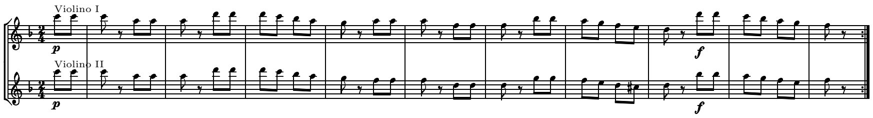
Using this representation, we have sketched a correct and appropriate basis for distributed protection which is proof-based and amenable to analysis. It addresses many of the open questions in distributed security including those of correctness, accountability and delegation, and has the potential to formally satisfy the apparently simple requirements for the higher security classifications of the Orange Book ([DoD85]) while retaining a very high expressiveness. The system is easily implementable at any level, and an efficient implementation may be constructed and deployed non-intrusively alongside or around existing systems.

A description of a non-distributed implementation of this system may be found in section 9.13.

Material which follows from this section may be found in:

- Section 12: The system presented here is only a sketch. There is considerable development possible both of the techniques used for building this system from the theoretical groundwork, and of this system itself, which satisfies a set of particularly valuable requirements. We suggest some possible approaches to continuing this development in section 12.

Why did we do all this . . . ?



11 Conclusions

We have developed and explored a full formalism for protection systems.

- **This formalism is powerful**, and has allowed us to produce many important and useful, if occasionally surprising computational results.
- **It is appropriate**, allows us to perform case studies and analyses of existing systems to identify their strengths and weaknesses.
- **It is practical**, and can be used to develop new protection systems with well-understood computational properties.
- **It is useful** since it may be used by systems administrators and analysts to audit real, hypothetical and proposed systems and configurations.

In the course of this development, we achieved a number of major technical results.

- We introduced the **current principal** and showed how it affects the safety problem. We used this to construct a **formal statement of the general safety problem**.
- We defined **simulation**, **equivalence** and **expressiveness** of protection systems.
- We constructed many **classes of protection system** and produced **decidability results** for the class safety problems for these classes.
- We described a menagerie of **practical protection systems** and exhibited their computational and organisational properties.

- We **unified the many mechanisms for the identification of the current principal within a common lambda calculus** and showed how they are related.
- We exhibited a **new computational model for principals**; based on our earlier work, this model maintains all the desirable properties of earlier models.

11.1 Safety

The first of the two core problems addressed by this thesis is that of safety: the problem of deciding whether a user is able to access an object directly or indirectly within a given protection system.

This problem is uninteresting without motivation, and a brief detour into the realms of policy provided us with a formal definition of 'adequate' or 'correct' protection, by which we motivated the safety problem.

In section 4, we reproduced the original formalism for a protection system from [HRU76], and corrected and extended it to include the concept of the 'current principal'. It was this model which we used as a basis for our analysis of safety in protection systems.

As we learned to derive computational properties of protection systems, it became increasingly necessary to compute with the systems themselves, to have equivalence and containment relations, and metrics of utility or expressiveness. While these were developed out of necessity in parallel with the proofs in section 6, they were all presented in section 5.

Section 6 demonstrates a number of surprising known and original results about the undecidability of the safety of remarkably restricted subclasses of protection system.

This plethora of undecidability results suggested that we approach the problem from another perspective: Instead of attempting to restrict an overgeneralised model, in section 7 we chose the simplest possible system and built up upon it until it became useful. Our work on practical protection systems completes the exploration of this particular class of systems. More complex systems exist, but they are not more expressive, and are likely to be of little more utility.

11.2 Responsibility

As early as section 2.1.3, we realised that protection consists of three stages, rather than two. Our correction in section 4 restored the missing stage, the 'identification of the current principal', to the formal model of a protection system.

The 'current principal' is defined to be the principal responsible for a particular operation. The problem of joint responsibility forced us to reevaluate our concept of the principal as a composite designation of responsibility, and section 8 contains a number of calculi for manipulating principals within the λ-calculus. Amongst the formal results of this section is an embedding of the stack inspection calculus from [FG02] into a data inspection calculus which does not suffer from the weaknesses of stack inspection systems and is capable of modelling the operation of tainting and capability systems. Tracking responsibility using this data inspection calculus makes the system invulnerable to any form of Trojan horse; this is the unique calculus with this property.

11.3 Application

The case studies in section 9 were short and sweet, largely as a consequence of the accuracy and relevance of our formal model. They were also largely disappointing, since it soon became clear quite how little of the concept of safety and the methods of constructing safe systems are understood by systems designers and builders. All of the systems studied suffer from major limitations or vulnerabilities; these are occasionally mitigated by either the implementation or the chosen application domain.

We sketched a case study of a new design in section 10. The system described in that section is almost a pure application of the formalisms developed in sections 7 and 8, and thus all the computational results known for those formalisms will hold for that system. We showed that this new system can be implemented with small overhead, using low order time and space.

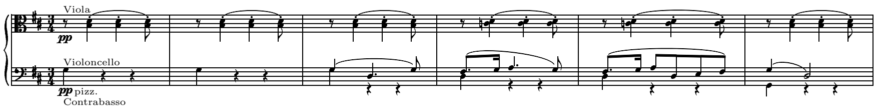
11.4 Closing Address

Any movement away from the current state of paranoia in system design requires considerable impetus: from the system designers who must build such the system; from the users, who must deploy it; and from the applications writers, who must take advantage of the features it has to offer. Currently, the applications writers are the most enthusiastic of these groups. Mobile code delivery systems including Java ([Sun03d]), Flash ([Mac03]) and ActiveX ([Mic03a]) and mobile games platforms like N-Gage ([Nok03]) would all benefit from better security models. However, established users with established business practices are rightfully wary of any "*new! secure! systems!*" which seem to be touted almost as regularly as 419 "advanced fee" scams.

Of all these groups, perhaps the systems builders are the quietest. Good designs are few and far between, and it is sometimes difficult to tell the difference between a good design and a bad design until an issue manifests. Systems have been known to survive for years before a fatal security flaw is discovered, and then suddenly, all hell breaks loose. If significant investment is to be placed into the implementation and deployment of a major new security model, it had better be a good one!

The road to a good security model is paved with formalisms.

You missed a bit ...



12 Further Work

In a field of this size, there is huge scope for further work. The list of suggestions here is woefully incomplete and disorganised, but should present some ideas for future directions.

- The author has designed a new and correct algorithm for the *implementation* of the data inspection calculus which does *not* have the overheads associated with traditional implementations. This algorithm should be described, developed and implemented.

- There is a possible definition of "*full expressiveness*" which neatly encapsulates the concept that a protection system must have sufficient expressiveness to achieve all required tasks, but must not permit any principal to be too expressive; that is, be unsafe. A protection system is "*fully expressive*" if a principal may create any [permitted] subprincipal of itself, but may create no principal wholly contained within itself. The word "*permitted*" covers any explicit restriction which may be imposed by policy. The '*ideal*' systems of section 7 are fully expressive for an arbitrary designation of '*permitted*' which may be expressed using the *grant* rights of the configuration. We did not have the vocabulary or understanding to express this definition in section 5 or use it in section 7 since we did not have the concept of containment of principals. However, it is by this definition that these systems may objectively be termed '*ideal*'.

- Perhaps the most obvious continuation would be to complete and deploy a practical implementation of the model described in section 10. This should be a relatively simple project from a technical standpoint, but doubtless there are many issues of implementation to be discovered.

  The actual design and implementation of any '*ideal protection system*' on a large scale presents a considerable technical and managerial challenge which would probably need to involve standards bodies as well as implementors.

- The relationship between the privilege sets of the λ_{sec}-calculus and the privilege sets identified by the graph walk of the ideal models merits considerable further study. A brief sketch of the utility of this relationship is made in section 10.1.2, but it is incomplete.

- It should be fairly simple to construct and prove theorems about a logic of responsibility equivalent to our λ_{sec}-calculus, but which models the computations of a parallel system. This is especially valuable in the network case where multiple processes or agents may be computing simultaneously on the same data.

- Our systems generally assume that information about the protection system is entirely public. However, there are many cases where exposing the hierarchy of principals within a company, for example, is not desirable. It seems quite feasible to establish conventions by which some parts of the hierarchy could be kept private, without affecting the computational nature or properties of the system. This work was introduced briefly in section 10.2.5.

- Given our understanding of the required interfaces between virtual machines, principals and reference monitors, it should be fairly simple to design an extension to the Java Native Interface ([Sun03c]) allowing a VM-independent security framework to be implemented in GNU Classpath ([GNU03]).

- The techniques for abstract interpretation introduced by [Ørb97] can be extended using the tristate logic of partial principals from section 8.10 to produce a useful and practical optimisation for language and compiler designers.

- There has been some work on security policy (for example [Ort96]), and it would be of considerable benefit to extend this work to understand which classes of policy may be satisfied by level k transitive systems. We believe this class to be quite large, almost certainly including all policies expressible in simple and negated predicates without boolean operators.

- The logic in section 5.4 was barely exposed and appears to have considerable potential. It may be extended to cover more situations. The cases where it is nonconservative should be identified. It may be possible to use this logic to give better than NP or EXPTIME results for monooperational systems.

- The class safety problem for biconditional monooperational protection systems appears especially worthy of study since so many of our ideal systems are in this class. It would be nice if the class safety problem were decidable in short time, but this does not seem likely.

- Grammatical systems (section 6.6 look interesting and are possibly a more general decidable class than simple transitive systems. Enough is known about languages and grammars that a study of grammatical classes should lead to a result.

- It may be possible to run an ideal system over a fully generalised representation of a principal as a set. In this case, algorithms must be designed for computing the privileges required to access the grant principal of any principal. This would probably be mathematically enlightening, but of less practical use since the artificial lattice of principals has no disadvantages over the fully general representation of principals as sets, as long as the logic of partial principals from section 8.10 is used.
- It may be possible to unify the artificial lattice of principals with existing certificate granting hierarchies. This seems unlikely, since certification hierarchies are designed around hierarchies of trust, rather than organisational structure. Certificate-based systems also suffer from problems of over-zealous revocation when a trust chain is broken.

What did $\sum \lambda_\nu x.\kappa(\Psi)$ mean again ...?

A Auxiliary Definitions and Proofs

A.1 Ordering

Definition A.1 (Ordered Set). An ordered set is a set S with a relation $<$ such that for all $a, b, c \in S$, the following hold:

1. Exactly one of $a = b$, $a < b$ or $b < a$ is true.

2. If $a < b$ and $b < c$ then $a < c$.

Definition A.2 (Partially Ordered Set). A partially ordered set is a set S with a relation $<$ such that for all $a, b, c \in S$, the following hold:

1. At most one of $a = b$, $a < b$ or $b < a$ is true.

2. If $a < b$ and $b < c$ then $a < c$.

A.2 The Schröder-Bernstein Theorem

Theorem (Schröder-Bernstein Theorem ([Cam99]), as theorem 5.3 on page 59). *If there is an injective map from X to Y and an injective map from Y to X, then there is a bijective function from X to Y. In other words, if $|X| \leq |Y|$ and $|Y| \leq |X|$ then $|X| = |Y|$.*

Proof. (Reproduced from [Cam99])

Let $f : X \to Y$ and $g : Y \to X$ be injective.

Without loss of generality, X and Y are disjoint. (Given X and Y, we can find X' and Y' bijective with X and Y, for example, take $X' = X \times \{1\}$ and $Y' = Y \times \{2\}$).

We say that $y \in Y$ is the *parent* of $x \in X$ if $g(y) = x$. $x \in X$ is the parent of $y \in Y$ if $f(x) = y$. Each element of X or Y has at most one parent. An *ancestral chain* for $z \in X \cup Y$ is a tuple $(z_0, z_1, \ldots, z_n$ such that $z_0 = z$ and z_{i+1} is the parent of z_i for each $i = 0, \ldots, n-1$. (Its elements belong alternately to X and Y.) The *length* of such a chain is n (the number of *steps*, not the number of *elements*).

There are two possibilities for any element z. Either there are arbitrarily long ancestral chains for z, in which case we shall say that z has *infinite depth*, or there is a unique longest ancestral chain for z ending with an element with no parent, in which case we say that the length of this chain is the *depth* of z. (The second possibility includes the case when z itself has no parent, in which case its depth is 0.) We let X_e denote the set of elements of X whose depth is even; X_o the set of elements of X with odd depth; and X_∞ the set of elements with infinite depth. We define Y_e, Y_o, Y_∞ similarly.

If $x \in X$ has finite depth, then $f(x)$ has depth one greater than the depth of X and if $x \in X$ has infinite depth, then so does $f(x)$. So f maps $X_e \to Y_o$, $X_o \to Y_e$ and $X_\infty \to Y_\infty$. A similar assertion holds for the action of g on elements in Y. Furthermore, elements of Y_o or Y_∞ have parents; so f maps $X_e \to Y_o$ and $X_\infty \to Y_\infty$ bijectively. (This does not hold for $X_o \to Y_e$ since an element of Y_e may have no parent.

Define a map $h : X \to Y$ by:

$$h(x) = \begin{cases} f(x) & \text{if} \quad x \in X_e \cup X_\infty \\ g^{-1}(x) & \text{if} \quad x \in X_o \end{cases}$$

Then it is easily seen that h is a bijection.

□

This proof requires some elementary properties of natural numbers which are not required for a second proof produced in [Cam99].

A.3 Bell numbers

Theorem A.1 (Bell Numbers). *The number of partitions of a set of n elements is given by the Bell number B_n. These may be defined recursively by*

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

where $B_0 = 1$ and $\binom{n}{k}$ is a binomial coefficient.

Proof. By induction: There is only one partition of an empty set, therefore $B_0 = 1$. Also $B_1 = 1$.

Let the number of possible partitions of a set containing n elements be B_n. We will count the partitions of a set containing $n+1$ elements. Consider the partition containing some chosen element, which we will call 1. Assume that it contains $n-k+1$ elements. There are $\binom{n}{k}$ ways to choose the remaining $n-k$ elements of this partition, leaving k elements. There are B_k ways to partition these remaining k elements. Thus, if the partition containing 1 is of size $n-k+1$, then there are $\binom{n}{k}B_k$ possible ways to partition the set of size n.

We let the value of $n-k+1$ range from 1 to $n+1$ and sum $\binom{n}{k}B_k$ over this range to get the given formula.

□

An alternative definition of the Bell numbers is given by

$$B_n = \sum_{k=0}^{n} S(n,k)$$

where $S(n,k)$ is the Stirling number of the second kind given by

$$S(n,k) = \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^n$$

The Stirling number $S(n,k)$ represents the number of ways a set of size n may be partitioned into k subsets. We also have a recurrence relation for the Stirling numbers given by

$$S(n,k) = S(n-1,k-1) + kS(n-1,k)$$

More information on Bell and Stirling numbers is available from [Wei].

A.4 Countability

Definition A.3 (Denumerability). A set A is "*denumerable*" if there exists a bijection between A and the natural numbers $\mathbb{N}$.

Definition A.4 (Countability). A set A is "*countable*" if it is denumerable or finite.

If we allow the axiom of choice, then it may be proven in a slightly heavier handed manner that any finite set is denumerable.

Theorem A.2 (Countability of the Integers). *$\mathbb{Z}$ is countable.*

Proof. Define $\phi : \mathbb{Z} \to \mathbb{N}$ by

$$\phi(n) = \begin{cases} 2n & \text{if } n \geq 0 \\ -2n-1 & \text{if } n < 0 \end{cases}$$

By inspection, ϕ is a bijection. Therefore $\mathbb{Z}$ is countable.

□

Theorem A.3 (Countability of the Positive Integers). *Let $\mathbb{N}_1 = \{1, 2, 3, \ldots\}$. Then $\mathbb{N}_1$ is countable.*

Proof. Define $\phi : \mathbb{N}_1 \to \mathbb{N}$ by

$$\phi(n) = n - 1$$

By inspection, ϕ is a bijection. Therefore $\mathbb{N}_1$ is countable.

□

Lemma A.1 (Conditions for Countability). *To show that a set A is countable, it is sufficient to find a bijection from A to any of $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{N}_1$.*

Theorem A.4 (Countability of Subsets). *Any subset of a countable set is countable.*

Proof. Let $A \subset B$ where B is countable. We have a bijection $\phi : B \to \mathbb{N}$, and a subset $N \subset \mathbb{N}$ such that a restriction of ϕ is a bijection $\phi\big|_A : A \to N$. It therefore suffices to show that any subset $N \subset \mathbb{N}$ is countable.

N has a total order and hence a least element n_0. Thus we may map this least element to 0. $N \backslash n_0$ has a least element n_1, which we map to 1, and so on.

□

It is therefore sufficient to find a bijection between a set A and a subset of $\mathbb{N}$ to prove countability, in other words, $\phi : A \to \mathbb{N}$ is required only to be an injective map.

Definition A.5 (Enumeration). An enumeration of a set A is an injective map $\kappa : A \to \mathbb{N}$ such that $\kappa : A \to \{1, 2, \ldots, B\}$ is a bijection and $\kappa^{-1}(n)$ is not defined for $n > B$.

An enumeration defines a particular total order on a countable set A. Each element may be considered to have a number, an index into the set starting from 1. Each element $a \in A$ may have at most one *next* element a_n such that $\kappa(a_n) = \kappa(a) + 1$, and at most one *previous* element a_p such that $\kappa(a_p) + 1 = \kappa(a)$. Any such nonempty set has exactly one *first* element with no previous element and exactly one *last* element with no next element.

Theorem A.5 (Countability of the Cartesian Product). *Let A and B be countable. Then $A \times B$ is countable.*

Proof. By hypothesis, there exist enumerations κ of A and χ of B. Let a_1 be the element of A such that $\kappa(a_1) = 1$, and so on to define $a_2, a_3, \ldots$; likewise for χ and B to define $b_1, b_2, b_3, \ldots$.

Consider the following function $\phi : A \times B \to \mathbb{N}$:

$$\phi((a_m, b_n)) = t(m + n - 2) + n$$

where $t(x)$ is the x^{th} triangular number given by

$$t(x) = \frac{x(x-1)}{2}$$

This counts diagonally along lines from the bottom left to the top right.

		n				
		1	2	3	4	5
	1	1	3	6	10	15
	2	2	5	9	14	
m	3	4	8	13		
	4	7	12			
	5	11				

□

Theorem A.6 (Countability of the Rationals). $\mathbb{Q}$ *is countable.*

Proof. $\mathbb{Q}$ is isomorphic to $\mathbb{N} \times \mathbb{Z}$ by definition.

□

Theorem A.7 (Countability of Multidimensional Spaces). $\mathbb{N}^n$ *is countable when n is finite.*

Proof. By induction: $\mathbb{N}^1$ is countable. If $\mathbb{N}^n$ is countable, then $\mathbb{N}^n \times \mathbb{N}$ is countable by theorem A.5. Therefore $\mathbb{N}^{n+1}$ is countable.

□

An alternative formulation of this theorem is, “Let A_i be countable for all $i \in \mathbb{N}$. Then $A_0 \times A_1 \times \ldots \times A_n$ is countable.” Any space isomorphic to $\mathbb{N}^n$ for some finite n is therefore countable.

Theorem A.8 (Uncountability of the Reals). *$\mathbb{R}$ is not countable.*

Proof. The simple version, by contradiction. Assume that L is a complete list of real numbers. Let L_0 represent the first item of the list L. We let x^i represent the i^{th} digit of x, and in particular, we let L_k^i represent the i^{th} digit of L_k. We can construct a new number x such that $x^0 \neq L_0^0$, $x^1 \neq L_1^1$, $x^2 \neq L_2^2$, etc. Now since x differs from each L_k by at least one digit, $x \notin L$. Thus L was not a complete list of the real numbers.

□

Theorem A.9 (Uncountability of the Power Set of an Infinite Set). *If a set A is infinite, then 2^A is uncountable.*

Proof. The proof is very similar to that of the uncountability of the real numbers in theorem A.8.

□

A.5 The Halting Problem

Theorem A.10 (The Halting Problem). *There is no effective procedure to determine whether an arbitrary Turing machine will halt on a given input.*

Proof. By contradiction. Assume that there exists a Turing machine H which reads a description of a Turing machine D from a tape and determines whether D would halt given its own description as input. In other words, $H(D)$ computes whether the function $D(D)$ halts. We modify H so that if $D(D)$ halts, $H(D)$ does not halt, and if $D(D)$ does not halt, $H(D)$ halts.

We now compute $H(H)$. If $H(D)$ halts, then $D(D)$ did not halt. But now $D = H$ so $H(H)$ was the program that did not halt. Equally, if $H(D)$ does not halt, then $D(D)$ did halt. But since $D = H$, then it was $H(H)$ that did halt.

Thus a contradiction is established. The machine H which decided the halting problem cannot exist.

□

A.6 Lattices

Definition A.6 (Lattice). A lattice is a structure $(A, \sqcap, \sqcup)$ with $\sqcap$ (the “*meet*”) and $\sqcup$ (the “*join*”) binary relations on A such that for all $x, y, z \in A$ the following six relations hold.

1	$x \sqcap y = y \sqcap x$	4	$x \sqcup y = y \sqcup x$
2	$x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$	5	$x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$
3	$x \sqcap (x \sqcup y) = x$	6	$x \sqcup (x \sqcap y) = x$

Bibliography

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 147–160, New York, NY, January 1999. ACM.

[Ano] Anonymous. The Jargon File: Back door. `http://www.jargon.net/jargonfile/b/backdoor.html`.

[Ano99] Anomymous. Summary about POSIX.1e, 1999. `http://wt.xpilot.org/publications/posix.1e/`.

[Ash02] Elaine Ashton. The perl timeline, 2002. `http://history.perl.org/PerlTimeline.html`.

[B+72] D. Bobrow et al. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM*, 15:135–143, 1972. Bolt, Beranek and Newman's TENEX.

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1981 (1st ed) revised 84.

[Bud83] Timothy A. Budd. Safety in grammatical protection systems. *International Journal of Computer and Information Sciences*, 12(6):413–431, December 1983.

[Byo98] Jon Byous. Java technology: An early history, 1998. `http://java.sun.com/features/1998/05/birthday.html`.

[Cam99] Peter J. Cameron. *Sets, Logic and Categories*. Springer Verlag, School of Mathematical Sciences, Queen Mary and Westfield College, London, E14NS, UK, 1999.

[Car60] Lewis Carroll. *The Annotated Alice*. Clarkson N. Potter, 1960. Alice's Adventures in Wonderland & Through the Looking-Glass, Edited by Martin Gardner.

[Cen88] National Computer Security Centre. *Glossary of Computer Security Terms (NCSC-TG-004)*, October 1988. The "Aqua Book".

[CGJ88] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. Performance effects of architectural complexity in the Intel 432. *ACM Transactions on Computer Systems*, 6(3):296–339, August 1988.

[Cha03] Chambers Harrap. *The Chambers 21st Century Dictionary*. Chambers Harrap Publishers Ltd, 2003.

[CJ75] E. Cohen and D. Jefferson. Protection in the Hydra Operating System. *ACM SIGOPS*, 9(5):141–160, November 1975.

[Com99] Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation*, August 1999. Version 2.1, adopted by ISO/IEC as ISO/IEC International Standard (IS) 15408 1-3. Available from `http://csrc.ncsl.nist.gov/cc/ccv20/ccv2list.htm`.

[Cop98] Crown Copyright. The Data Protection Act, 1998. `http://www.legislation.hmso.gov.uk/acts/acts1998/19980029.htm`.

[Cri65] P. A. Crisman, editor. *The Compatible Time-Sharing System.* M.I.T. Press, Cambridge, Mass., 1965.

[Dal03] Dalnet Administration. The DALnet FAQ, 2003. `http://help.dal.net/faqs/dalnetfaq.html`.

[DD93] C. J. Date and Hugh Darwen. *A Guide to the Sql Standard.* Addison-Wesley, Reading, Mass., 3 edition, 1993.

[Den76] P. J. Denning. Fault tolerant operating systems. *Computing Surveys (USA)*, 8(4):359–389, December 1976.

[DoD85] US Department of Defense. Trusted computer system evaluation criteria. Technical Report 5200.28, US Department of Defense, 1985. The “Orange Book”.

[DvH66] J. B. Dennis and E. C. van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[F$^+$03] Philip Fong et al. AegisVM, 2003. `http://aegisvm.sourceforge.net/`.

[FG02] Cédric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. *ACM SIGPLAN Notices*, 31(1):307–318, January 2002.

[FL94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In USENIX, editor, *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 97–114, Berkeley, CA, USA, 1994. USENIX.

[Fla96] David Flanagan. *Java in a Nutshell, First Edition.* O’Reilly and Associates, Inc, 1996.

[Fla98] David Flanagan. *JavaScript: The Definitive Guide.* O’Reilly and Associates, Inc, 1998.

[Fla99] David Flanagan. *Java in a Nutshell, Third Edition.* O’Reilly and Associates, Inc, 1999.

[Fou04] The Perl Foundation. The perl directory, 2004. `http://www.perl.org/`.

[FS96] N. Ferguson and B. Schneier. *Practical Cryptography.* John Wiley & Sons, Inc, New York, NY, 1st edition, 1996.

[fS03] International Organization for Standardization. *ISO/IEC 9075-2:2003: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation).* International Organization for Standardization, Geneva, Switzerland, 2003.

[GNU03] GNU developers, see the AUTHORS file in the distribution for a complete list. GNU Classpath, 2003. `http://www.gnu.org/software/classpath/`.

[Gon03] Li Gong. The Java Security Architecture, 2003. `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.d%oc.html`.

[Gor03] Thomas F. Gordon. Models and Software for eGovernance, 2003.

[Gur89] Eitan M. Gurari. *An Introduction to the Theory of Computation.* Computer Science Press, Ohio State University, 1989.

[Han04] Ask Bjørn Hansen. Parrot, 2004. `http://www.parrotcode.org/`.

[Har85] Norm Hardy. The KeyKOS architecture. *ACM Operating Systems Review*, 19(9):8–25, September 1985.

[Har88] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review, SIGOPS*, 22(4):36–38, October 1988.

[Har97] Elliotte R. Harold. The comp.lang.java faq, 1997. `http://www.ibiblio.org/javafaq/javafaq.html`, `http://www.cafeaulait.org/javafaq.html`.

[Hie01] Jarkko Hietaniemi. Perl Programmers' Reference Guide: PERLHIST, 2001.

[Hon70] Honeywell. *Multics Level 68 Programmers' Manual Reference*, undated (circa 1970).

[HR78] M. Harrison and W. Ruzzo. Monotonic protection systems. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 337–365. Academic Press, New York, 1978.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge, UK, 1986. London Mathematical Society Student Texts 1.

[IBM03] IBM. The Jikes Research Virtual Machine (RVM), 2003. A flexible open testbed for prototyping virtual machine technologies, available at `http://oss.software.ibm.com/developerworks/oss/jikesrvm/`.

[IG03] Intel and GNU developers. Open Runtime Platform, an open source research platform for java, 2003. `http://orp.sourceforge.net/`.

[Inc04] O'Reilly Media Inc. Perl.com: The source for perl, 2004. `http://www.perl.com/`.

[IOS00] International Organization for Standardization. *ISO/IEC 17799:2000: Information technology — Code of practice for information security management.* International Organization for Standardization, Geneva, Switzerland, 2000. Also available from British Standards as BS7799-1:2000.

[JLS76] A. K. Jones, R. J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *Proc. 17th Annual Symp. on Foundations of Computer Science*, 1976.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Inc, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[KO04] The Kernel.Org Organization. The linux kernel archives, January 2004. `http://www.kernel.org/`.

[Lam71] Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[LB78] Richard J. Lipton and Timothy A. Budd. On classes of protection systems. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 281–296, New York, 1978. Academic Press.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[LS76] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, May 1976.

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, July 1977.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1997.

[Mac03] Macromedia. Flash MX 2004, 2003. `http://www.macromedia.com/software/flash/`.

[Man97] Ben Mankin. The Anarres II MUD, 1997.

[Man03] Ben Mankin. The Anarres II Documentation Project, 2003.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[McK99] Marshall Kirk McKusick. Twenty Years of Berkeley Unix: From AT&T-Owned to Freely Redistributable, January 1999. `http://www.oreilly.com/catalog/opensources/book/kirkmck.html`.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine.* O'Reilly and Associates, Inc, 1997.

[Mic03a] Microsoft. ActiveX Controls - Microsoft Papers, Presentations, Web Sites and Books, 2003. http://www.microsoft.com/com/tech/ActiveX.asp.

[Mic03b] Microsoft. Microsoft .NET, 2003. http://www.microsoft.com/net/.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 66 Wood Lane End, Hemel Hempstead, Hertfordshire, HP2 4RG, 1989.

[Min72] Marvin Minsky. *Computation: Finite and infinite machines.* Prentice-Hall, London, 1972.

[Min78] Naftaly Minsky. The principle of attenuation of privileges and its ramifications. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 337–365, New York, 1978. Academic Press.

[Mof03] Nick Moffitt. Nick moffitt's $7 history of unix, October 2003. http://www.crackmonkey.org/unix.html.

[MS96] Yuri Matiyasevich and Géraud Sénizergues. Decision problems for semi-Thue systems with a few rules. In *Proceedings, 11*$^{\text{th}}$ *Annual IEEE Symposium on Logic in Computer Science*, pages 523–531, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[MvOV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1st edition, 1996.

[MW93] John-Jules Ch. Meyer and Roel J. Wieringa, editors. *Deontic Logic in Computer Science.* Wiley Professional Computing Series. John Wiley and Sons, Chichester, UK, 1993.

[MyS03] MySQL AB. How the privilege system works, 2003. http://www.mysql.com/doc/en/Privileges.html.

[NCC99] Netscape Communications Corporation. *Client Side JavaScript Guide for version 1.3.* Netscape Communications Corporation, 466 Ellis Street, Mountain View, CA 94043-4042, 1999. Available at http://devedge.netscape.com/library/manuals/2000/javascript/1.3/guide/ Notes on security have mysteriously vanished from the documentation for later versions 1.4 and 1.5.

[NH82] R. M. Needham and A. J. Herbert. *The Cambridge CAP Computer and Its Operating System.* Addison-Wesley, Reading, MA, 1982.

[Nok03] Nokia. N-gage home, 2003. http://www.n-gage.com/.

[NW74] R. M. Needham and R. D. M. Walker. Protection and process management in the CAP computer. In *Proc. IRIA Internat'l Workshop on Protection in Operating Systems*, pages 155–160, Rocquencourt, France, 1974. Institute de Recherche d'Informatique et d'Automatique, France.

[NWG82] Network Working Group. RFC 819: The Domain Naming Convention for Internet User Applications, August 1982.

[NWG93] Network Working Group. RFC 1459: Internet Relay Chat Protocol, May 1993.

[NWG03] Network Working Group. RFC 3530: Network File Systems (NFS) version 4 Protocol, April 2003.

[OG03] The Open Group. The Single Unix Specification, 2003. `http://www.unix.org/single_unix_specification/`.

[Ørb95] Peter Ørbaek. Can you trust your data? In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 575–589. Springer-Verlag, 1995.

[Ørb97] Peter Ørbæk. *Trust and Dependence Analysis.* PhD thesis, Dept. of Computer Science, University of Aarhus, 1997.

[Org73] E. Organick. *Computer System Organization: The B5700/B6700 Series.* Academic Press, 1973.

[Ort96] Rodolphe Ortalo. Using deontic logic for security policy specification, 1996.

[PASC97] Portable Applications Standards Committee of the IEEE Computer Society. *Draft Standard for Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) Amendment (?): Protection, Audit and Control Interfaces [C Language].* The Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY10017, USA, 1997. IEEE Std 1003.1e.

[Per01] Perl Programmers Reference Guide: PERLSEC, 2001.

[Pfl97] C. P. Pfleeger. *Security in Computing.* Prentice Hall International, 2nd edition, 1997.

[PGDG03] The PostgreSQL Global Development Group. PostgreSQL Documentation: GRANT, 2003. `http://developer.postgresql.org/docs/postgres/sql-grant.html`.

[Pos46] E. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 53:264–268, 1946.

[Pra03] Henry Prakken. (untitled home page), 2003. `http://www.cs.uu.nl/people/henry/`.

[PSS01] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001.

[RHFL86] S. A. Rajunas, A. C. Hardy, N. Bomberger, W. S. Frantz, and C. R. Landau. Security in keyKOS. In *Proc. IEEE Symposium on Security and Privacy*, pages 78–85, 1986.

[Rit79] Dennis Ritchie. The evolution of the unix time sharing system, circa 1979. `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html`.

[RYK02] George Reese, Randy Jay Yarger, and Tim King. *Managing and Using MySQL*. O'Reilly and Associates, Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2002.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[Sch96] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc, New York, NY, 2nd edition, 1996.

[Sha99] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, April 1999.

[Shi00] R. Shirey. FYI 36: Internet security glossary, May 2000.

[SK03] Eugene Spafford and Gene Kim. Tripwire open source project, 2003. `http://www.tripwire.org/`.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke University Press, 1996.

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 170–185. Kiawah Island Resort, near Charleston, South Carolina, December 1999.

[Sto70] D. Stone. Pdp-10 system concepts and capabilities. *PDP-10 Applications in Science*, II:32–55, undated (ca. 1970). Digital Equipment Corporation's DECsystem/10.

[Sun96] Sun Microsystems. The Java Development Kit class source code, version 1.0.2, 1996. `http://java.sun.com/products/archive/`.

[Sun03a] Sun Microsystems. The Java 2 API documentation, 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[Sun03b] Sun Microsystems. The Java 2 SDK standard edition source code, version 1.4.1, 2003. `http://java.sun.com/j2se/1.4.1/`.

[Sun03c] Sun Microsystems. The JNI FAQ, 2003. `http://java.sun.com/products/jdk/faq/jnifaq.html`.

[Sun03d] Sun Microsystems. The source for Java Technology, 2003. http://java.sun.com/.

[TCOS90] IEEE Technical Committee on Operating Systems and Application Environments. *Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language].* The Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY10017, USA, 1990. ISO/IEC 9945-1 : 1990 / IEEE Std 1003.1.

[Tec02] Lucent Technologies. The Creation of the Unix Operating System, 2002. http://www.bell-labs.com/history/unix/.

[Tho84] K Thomson. Reflections on trusting trust. *Communications of the A.C.M.*, 27(8), August 1984.

[TVV03] Multicians, 2003. http://www.multicians.org/.

[vdL89] Rick F. van der Lans. *The SQL standard: a complete reference.* Academic Service, Postbus 81, 2870 AB Schoonhoven, The Netherlands, 1989. Translated from the Dutch.

[Wal99] Dan S. Wallach. *A New Approach to Mobile Code Security.* PhD thesis, Princeton University, Department of Computer Science, January 1999.

[WCC$^+$74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operatng system. *Communications of the Association of Computing Machinery*, 17(6):337–45, June 1974.

[Wei] Eric W. Weisstein. Bell numbers. http://mathworld.wolfram.com/BellNumber.html.

[Wei69] C. Weissmann. Security controls in the adept-50 time-sharing system. *FJCC, AFIPS Conf. Proc.*, 35:119–133, 1969. System Development Corporation's Advanced Development Prototype (ADEPT).

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.

Index

“Begin at the beginning, the King said gravely, and go on till you come to the end: then stop.”
— Lewis Carroll, ‘Alice in Wonderland’