Policy-Directed Code Safety

by

David E. Evans

S.B. Massachusetts Institute of Technology (1994)
S.M. Massachusetts Institute of Technology (1994)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
at the
Massachusetts Institute of Technology

February 2000

Author……………………………………………………………………………………
David Evans
Department of Electrical Engineering and Computer Science
October 19, 1999

Certified by……………………………………………………………………………
John V. Guttag
Professor, Computer Science
Thesis Supervisor

Accepted by…………………………………………………………………...……………
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Policy-Directed Code Safety

by

David E. Evans

Submitted to the Department of Electrical Engineering and Computer Science in partial
fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Executing code can be dangerous. This thesis describes a scheme for protecting the user by constraining the behavior of an executing program. We introduce Naccio, a general architecture for constraining the behavior of program executions. Naccio consists of languages for defining safety policies in a platform-independent way and a system architecture for enforcing those policies on executions by transforming programs. Prototype implementations of Naccio have been built that enforce policies on JavaVM classes and Win32 executables.

Naccio addresses two weaknesses of current code safety systems. One problem is that current systems cannot enforce policies with sufficient precision. For example, a system such as the Java sandbox cannot enforce a policy that limits the rate at which data is sent over the network without denying network use altogether since there are no safety checks associated with sending data. The problem is more fundamental than simply the choices about which safety checks to provide. The system designers were hamstrung into providing only a limited number of checks by a design that incurs the cost of a safety check regardless of whether it matters to the policy in effect. Because Naccio statically analyzes and compiles a policy, it can support safety checks associated with any resource manipulation, yet the costs of a safety check are incurred only when the check is relevant.

Another problem with current code safety systems is that policies are defined in *ad hoc* and platform-specific ways. The author of a safety policy needs to know low-level details about a particular platform and once a safety policy has been developed and tested it cannot easily be transferred to a different platform. Naccio provides a platform-independent way of defining safety policies in terms of abstract resources. Safety policies are described by writing code fragments that account for and constrain resource manipulations. Resources are described using abstract objects with operations that correspond to manipulations of the corresponding system resource. A platform interface provides an operational specification of how system calls affect resources. This enables safety policies to be described in a platform-independent way and isolates most of the complexity of the system.

This thesis motivates and describes the design of Naccio, demonstrates how a large class of safety policies can be defined, and evaluates results from our experience with the prototype implementations.

Thesis Supervisor: John V. Guttag
Title: Professor, Computer Science

Acknowledgements

John Guttag is that rare advisor who has the ability to direct you to see the big picture when you are mired details and to get you to focus when you are distracted by irrelevancies. John has been my mentor throughout my graduate career, and there is no doubt that I wouldn't be finishing this thesis this millennium without his guidance.

As my readers, John Chapin and Daniel Jackson were helpful from the early proposal stages until the final revisions. Both clarified important technical issues, gave me ideas about how to improve the presentation, and provided copious comments on drafts of this thesis.

Andrew Twyman designed and implemented Naccio/Win32. His experience building Naccio/Win32 helped clarify and develop many of the ideas in this thesis, and his insights were a significant contribution to this thesis.

During my time at MIT, I've at the good fortune to work with many interesting and creative people. The MIT Laboratory for Computer Science and the Software Devices and Systems group provided a pleasant and dynamic research environment. Much of what I learned as a grad student was through spontaneous discussions with William Adjie-Winoto, John Ankcorn, Anna Chefter, Dorothy Curtis, Stephen Garland, Angelika Leeb, Ulana Legedza, Li-wei Lehman, Victor Luchangco, Andrew Myers, Anna Pogosyants, Bodhi Priyantha, Hariharan Rahul, Michael Saginaw, Raymie Stata, Yang Meng Tan, Van Van, David Wetherall, and Charles Yang. This work has also benefited from discussions with Úlfar Erlingsson and Fred Schneider from Cornell, Raju Pandey from UC Davis, Dan Wallach from Rice University, Mike Reiter from Lucent Bell Laboratories, and David Bantz from IBM Research.

Geoff Cohen wrote the JOIE toolkit used as Naccio/JavaVM's transformation engine and made its source code available to the research community. He provided quick answers to all my questions about using and modifying JOIE.

Finally, I thank my parents for their constant encouragement and support. I couldn't ask for two better role models.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional computer security has focused on assuring confidentiality, integrity and availability. Confidentiality means hiding information from unauthorized users; integrity means preventing unauthorized modifications of data; and availability means preventing an attacker from making a resource unavailable to legitimate users. Military and large commercial systems operators are (or at least should be) willing to spend large amounts of effort and money as well as to risk inconveniencing their users in order to provide satisfactory confidentiality, integrity and availability assurances.

The security concerns for typical home and non-critical business users are very different. In the past, these users had limited security concerns. Since they were typically not connected to a network, their primary concern was viruses on software distributed on floppy disks. Although viruses could be a considerable annoyance, users who stuck to shrink wrapped software were unlikely to encounter viruses, and the damage was limited to destroying files (or occasionally hardware) on a single machine.

Today, nearly all computers are connected to the public Internet much of the time. Although the benefits of connectivity are unquestioned, being on a network introduces significant new security risks. The damage a program can do is no longer limited to damaging local data or hardware – it can send personal information through the global Internet, damaging the operator’s reputation or finances. Furthermore, the likelihood of executing an untrustworthy program is dramatically increased. The ease of distributing code on the Internet means users often have little or no knowledge about the origin of the code they choose to run. In addition, it is becoming hard to distinguish the “programs” from the “data” – Java applets embedded in web pages can run unbeknownst to the user; documents can contain macros that access the file system and network; and email messages can contain attachments that are arbitrary executables.

The solution in high security environments is to turn off all mobile code and only run validated programs from trusted sources. This can be done by configuring browsers and other applications to disallow active contents such as Java applets and macros, or by installing a firewall that monitors all network traffic and drops packets that may contain untrustworthy code. This solution sacrifices the convenience and utility of the network, and would be unacceptable in many environments. Instead, solutions should allow possibly untrustworthy programs to run, but allow the user to place precise limits on what they may do. In such an environment, security mechanisms must be inexpensive and unobtrusive. Anecdotal evidence suggests that any code safety system that places a burden on its users will be quickly disabled, since its benefits are only apparent in the extraordinary cases in which a program is behaving dangerously.

A code safety system provides confidence that a program execution will not do certain undesirable things. Although much progress has been made toward this goal in the last few years, current systems are still unsatisfactory. This work seeks to address two important weaknesses of existing code safety systems:

1. They cannot enforce sufficiently precise policies. This means either a program is allowed to do harmful things, or users are unable to run some useful programs. For example, a system like the Java sandbox cannot enforce a policy that limits the number of bytes that may be written to the file system without preventing writing completely. This is a result of the limited locations where safety checking can be done. The designers were forced to select a small number of security-relevant operations that can have safety checking since the overhead of a safety check is always suffered even if the policy in effect places no constraints on the security-relevant operation.
2. The mechanisms they provide for defining safety policies are *ad hoc* and platform-specific. *Ad hoc* policy definition mechanisms limit the policies that can be defined to the class of policies considered by the system designers. It is impossible to anticipate all possible attacks or security requirements, so *ad hoc* policy definition mechanisms are inevitably vulnerable to new attacks. Tying policy definition to a particular execution platform means that policy authors need to know intimate details about that platform, and there is no opportunity to reuse policies on different execution platforms. This is a problem for policy authors, but also limits what policies are available to users. Further, it increases the gap between those people capable of writing and understanding policies and those who must trust a provided definition.

This thesis demonstrates that it is possible to produce a code safety system that does not suffer from these weaknesses without sacrificing convenience or efficiency. We describe Naccio[1], an architecture for code safety, and report on two prototype implementations: Naccio/JavaVM that enforces policies on JavaVM classes, and Naccio/Win32 that enforces policies on Win32 executables. Naccio defines policies by associating checking code with abstract resource manipulations. A Naccio implementation includes an operational specification of an execution platform in terms of those abstract resource manipulations. Naccio enforces policies by transforming programs to interpose checking code around security-critical operations.

1.1 Threats and Countermeasures

No security system can prevent all types of threats. Our focus is on threats stemming from executing programs. We ignore threats that do not result from a legitimate user running a program including compromised authentications and physical security breeches.

Different kinds of threats call for different countermeasures. Countermeasures for threats related to program executions come in two basic forms: restrictions on which programs may run, and constraints on what executions may do. Restrictions on which programs may run can be based on trust and cryptography (only run programs that are cryptographically signed by someone I trust), or based on static analysis that proves a program does not have certain undesired properties (only run programs that a virus detector checks do not contain instruction sequences matching known viruses). Constraints on what executions may do can be expressed as a policy.[2] The policy that

---

[1] The name Naccio is derived from *catenaccio*, a style of soccer defense popularized by Inter Milan in the 1960s. Catenaccio sought to protect the Inter net from attacks, by wrapping potential threats with a marker that monitors their activity and aggressively removing potentially dangerous parts (that is, the ball) from attackers as soon as they cross the domain protection boundary (also known as the midfield line).

[2] Not to be confused with an organizational security policy that specifies what policy to enforce on different types of programs.

should be enforced on an execution depends on how much trust the user has in the program and how much knowledge is available about its expected behavior. Ideally, all executions would run with a policy that limits them to exactly the behavior deemed acceptable for that program. This is not possible, however, since users cannot be expected to research and encode the limits of expected behavior for every program before running it. Instead, we should use different policies as countermeasures to different types of threats. Threats where code safety is an important countermeasure include viruses, Trojan horses, faulty programs and user mistakes.

Viruses

Viruses are code fragments that propagate themselves automatically. The damage they cause ranges from causing a minor annoyance to destroying hard drives and distributing confidential information. Every few weeks a new virus attack is reported widely in the mainstream media [NYTimes99a, NYTimes99b, NYTimes99c].

Although early computer viruses spread by attaching themselves to programs, extensibility features in modern email programs and web browsers make creating and spreading viruses much easier. A recent example is the Melissa Word macro virus [Pethia99]. It propagates using an infected Word document contained in an email message. When a user opens the infected document, the macro executes automatically (unless Word macros are disabled). The macro then lowers the macro security settings to permit all macros to run when future documents are opened and propagates itself by sending infected email messages to addresses found in the user's Microsoft Outlook address books. The macro also infects the standard document template file that is loaded by default by all Word documents. If the user opens another Word document, that document will be mailed along with the virus to addresses in the user's address books.

The most common virus countermeasures are virus detection programs such as McAfee VirusScan [McAfee99] and Symantec Norton AntiVirus [Symantec98]. Nearly every new PC comes with virus detection software installed. Most virus detectors scan files for signatures matching a database of known viruses. Commercial products for detecting viruses recognize tens of thousands of known viruses, and their vendors employ large staffs to identify new viruses.

The problem with this approach is that it depends on recognizing a known virus, so it offers no protection against new viruses. Because viruses like the Melissa macro virus can spread remarkably quickly over the Internet, they can do considerable damage before they are identified and virus detection databases can be updated. The damage inflicted by Melissa was limited to propagating itself and sending possibly confidential files to known addresses. A terrorist motivated to cause as much damage as possible could fairly easily create a variant of Melissa that inflicts far more harm.

To detect or prevent damage from previously unidentified viruses requires an approach that does not depend on recognizing a known sequence of instructions. Some commercial virus detection products include heuristics for identifying likely viruses based on static properties of the code or dynamic properties of an execution [Symantec99]. These approaches lead to an arms race between virus creators and virus detectors, as virus creators go to greater lengths to make their viruses hard to detect. Although heuristic detection techniques show some promise, it is unlikely that they will ever be able to correctly distinguish all viruses from legitimate programs.

A different approach is to limit the damage viruses can cause and their ability to propagate by observing and constraining program behavior. For example, the damage done by macro viruses could be limited by enforcing a policy on Microsoft Word executions. We would want to enforce different policies on Word executions depending on whether they were started to read a document embedded in an email message or web page, or started to edit a trusted document. When Word is used to edit a local document, perhaps a policy that prohibits any network transmission would be adequate. For documents from untrustworthy sources, a reasonable policy would require explicit permission from the user before Word transmits anything over the Internet, reads sensitive files, alters the registry, or modifies the standard document templates.

Trojan horses

A Trojan horse is an apparently useful program that also does some things the user considers undesirable. There have been many instances where an attacker has distributed a deliberately malicious program in the guise of a useful one. For example, someone distributed a version of `linux-util` that contained a `login` program that would allow unauthorized users to execute arbitrary commands [CERT99b].

In addition, there are programs a user may consider malicious even if the author did not intend to produce a malicious attack. For example, an early version of the Microsoft Network client would read and transmit the user's directory structure [Risks95]. While most users would be unaware that this is occurring, and would not be overtly damaged by it (other than losing bandwidth that could have been used for transmitting useful data), many would consider it a privacy violation.

Countermeasures for Trojan horses are similar to those for viruses, except that more precise policies may be needed. Although it would be difficult to monitor the information sent over the network by the Microsoft Network client, it would be possible to detect suspicious transmissions and alert the user. A more reasonable policy would ignore the actual transmitted data but place restrictions on which files, directories and registry entries could be examined, thereby limiting the information available to the program.

Faulty programs

Program bugs pose two different kinds of security threats – an attacker may deliberately exploit them or they may accidentally cause harm directly. The security advisories recorded by CERT [CERT99a] are rife with examples of buggy programs leading to exploitable security vulnerabilities. Of the 71 advisories posted between January 1996 and May 1999, 60 are directly attributable to specific program bugs (of these, 13 are the direct result of buffer overflows). A particularly vulnerable program is `sendmail`. Attackers have exploited various bugs in `sendmail` to gain root access [CERT96a, CERT96b], execute programs with group permissions of another user [CERT96c], and to execute arbitrary commands with root privileges [CERT97].

Other program bugs cause harm unintentionally. One notorious example is the Therac-25, a device for administering radiation to cancer patients [Leveson93]. Because of software bugs, it would occasionally administer a lethal dose of radiation and several patients died as a result. Although the system software had *ad hoc* safety checks, they were obviously not sufficient.[3] Because they were *ad hoc,* operators and doctors could not examine them and decide if the device was trustworthy.

The best way to obtain protection from exploitable or harmful program bugs would be to produce bug-free programs. Despite progress in software development and validation techniques, it is

---

[3] The Therac-25 disaster was the result of numerous factors ranging from flawed hardware design to poor regulation procedures. Although code safety mechanisms could be part of the solution, designing safety-critical systems involves far more than just code safety.

inconceivable that this will be accomplished in the foreseeable future. Since programs will inevitably contain bugs, code safety systems should be used to limit the damage resulting from buggy programs.

As with Trojan horses, the expected behavior of the program is known so it is reasonable to enforce a precise policy that limits what it can do. The difference is that the software vendor should be an ally in protecting the user from bugs, unlike a malicious attack. Security-conscious software vendors could include policies with their software distributions or even distribute their software with an integrated safety policy enforced. Reputable vendors should be motivated to protect their users from damaging bugs and might be expected to devote some effort towards producing a suitable policy. By separating the policy enforcement mechanisms from the application, they can have more confidence that the policy is enforced correctly. In addition, publishing an application's safety policy in a standard, easily understood format would give potential customers a chance to decide if the application is trustworthy.

User mistakes

Perhaps the most common way programs cause harm is unintentional mistakes by users. Because of poor interfaces or ignorance, users may inadvertently destroy valuable data or unknowingly transmit private information. One example is when an unsuspecting user issues the command `tar cf *` to create a new directory archive. This command will replace the contents of the first file in the directory with an archive of all other files, destroying whatever happened to be the first file. Although the program is behaving correctly according to its documentation, this is probably not the behavior the user indented. A well-designed interface lessens the risk of harmful user mistakes, but combining this with a user-selected and independently enforced policy is a more robust solution.

1.2 Background

Researchers have been working on limiting what programs can do since the early days of computing. Early work on computer security focused on multi-user operating systems built around a privileged kernel. The kernel is the only part of the system that manipulates resources directly. User programs must call functions in the operating system kernel to manipulate resources. The operating system limits what user programs can do to system resources by exposing a narrow interface and putting checks in the system calls to disallow unsafe resource use. Each application process runs in a separate address space, enforced by hardware support for virtual memory. A process cannot see or modify memory used by another process since it is not part of its virtual address space.

The problem with using separate processes to protect memory is that the cost of creating and maintaining a process is high, as is the cost of communicating and sharing data between processes. Switching between different processes involves a context switch, which is usually expensive. Several systems have attempted to provide the isolation offered by separate processes within a single process by using software mechanisms. We use *low-level code safety* to refer to security designed to isolate programs and require that all resource manipulations go through well-defined interfaces. It includes the control flow safety, memory safety, and stack safety needed to prevent programs from accessing arbitrary memory segments [Kozen98]. There are several ways to provide low-level code safety. Approaches such as the Java byte code verifier and proof-carrying code techniques statically verify that the necessary properties are satisfied. Software fault isolation provides the necessary guarantees by inserting masking or checking instructions to limit the targets of jumps and memory instructions. Section 7.1 describes work in low-level code safety.

Although Naccio depends on low-level code safety for the integrity of its policy enforcement mechanisms, the focus of this thesis is on policy-directed code safety. Policy-directed code safety seeks to enforce different policies on different executions. This can be done either by statically verifying the desired properties always hold, or by enforcing properties using run-time checking. Since it is infeasible to verify most interesting properties on arbitrary programs, most work has focused on run-time enforcement.

Most run-time constraint mechanisms, including Naccio, can be viewed as *reference monitors* [Lampson71, Anderson72]. A reference monitor is a system component that enforces constraints on access and manipulation of a resource. It should be invoked whenever the monitored resource manipulation occurs, and it should be protected from program code in a way that prevents bypassing or tampering. Reference monitor systems differ in how the monitors are invoked. They could be called explicitly by the operating system kernel, called by a separate watchdog process, or integrated directly into program code. Naccio integrates reference monitors directly into code, but takes advantage of system library interfaces to limit the code that must be altered.

Reference monitors also differ in how checking code is defined. Some possibilities include access matrices, finite automata, or general code. In a reference monitor security system, policies are limited by where reference monitor calls can be placed and what system state they may observe. There is usually a tradeoff between supporting a large class of policies and the performance and complexity of the system. Naccio security is based on reference monitors that can be flexibly introduced into programs at different points. This allows for a large class of policies to be enforced, but avoids the overhead necessary to support many reference monitors when a simple policy is enforced.

One example of a reference monitor is the `SecurityManager` used for high-level code safety in the Java virtual machine. API functions limit what programs can do by using the `SecurityManager` class. It acts as a reference monitor, enforcing a particular security policy by controlling access to system calls. The Java approach limits the policies that can be enforced since the only places reference monitors can be invoked are those defined as check methods in the `SecurityManager`. Developers can write a `SecurityManager` subclass that performs the desired checking for the given check methods, but cannot change the places where the API routines call check methods. For instance, the constructor for `FileOutputStream` calls the `SecurityManager.checkWrite` method before opening a file, but the `write` method that writes bytes to an open file does not check any `SecurityManager` method. Hence, one can implement an arbitrary security policy on what files may be written by writing code for the `checkWrite` method, but can place no constraints on the amount of data that may be written to a file once it has been opened. Other reference monitor systems are described in Section 7.3.

1.3 Design Goals

Naccio is intended to be a code safety system suitable for users in low and medium security environments. Although its mechanisms should be reliable enough for use in a high security environment, users in high-security environments should avoid untrustworthy code and rely on redundant mechanisms to avoid disasters. Further, high security users are willing to accept more obtrusive code safety mechanisms than would be acceptable in a less security-critical environment. Naccio could be useful as one of the pieces in a security system for a high-security environment, but would not be sufficient on its own.

We consider a low security user to be someone who is unsophisticated in security matters and who uses the Internet for web browsing and email. Low security users occasionally conduct transactions using the Internet and send and receive business-related email, but are not using their computer as a critical part of a business. The vast majority of current Internet users fit into this category. Medium security users are somewhat more sophisticated regarding security and have more to lose if there is a security breach. This category includes people running servers for small businesses and those with a substantial stake in their on-line reputation.

Some contexts where Naccio should be useful include:

- Executing remote code such as Java applets or ActiveX controls in a web browser. Typical low-security users allow their browser to run ActiveX controls with no constraints and Java applets with default constraints on what files may be read and written and what network connections may be made. This is reasonably acceptable today, since the damage an attacker could inflict on a typical user is low. This is changing though, and will continue to worsen as even typical users increasingly have a substantial stake in their on-line identity and store financial and personal information on their computer. Most medium-security users today configure their browser to disable ActiveX controls and either disable Java applets or allow them to run but worry that existing security measures are inadequate. While disabling remote code addresses the security issues, it sacrifices some of the richness of the web. More precise policies that can constrain a greater range of behavior should allow medium-security users to comfortably run remote code with assurances that it will not exhibit harmful behavior.
- Executing code in mail attachments. Most modern email programs support attachments that may be data files containing executable code (such as a Microsoft Word document) or a plain executable file. Two well-publicized recent attacks propagated using email attachments – the Melissa macro virus [Pethia99] propagates using a Word document attached to an email message and the Worm.ExploreZip virus [Cnet99a] propagates by attaching an executable file to an email message. Until these scares were widely publicized, typical low-security users would run mail attachments without reservations. Today, most are at least aware of the risks and will be reluctant to run attachments in messages coming from untrusted sources. Since the viruses mentioned above appear to be sent by people the user knows, however, this is not sufficient protection. A code safety system could solve the problem by allowing attachments to run, but enforce a policy that places constraints on their behavior.
- Uploadable code. Consider an auction site operator who wants to support programs submitted by clients that can access the server database, do some computation, place bids on behalf of its owner, and send messages to its owner. The site operator needs to limit the behavior of the client program including what files it can access and what network connections it may open, as well as place bounds on the server resources it may consume such as network bandwidth and database connections. Support for uploadable code is one of the largely unsatisfied promises of the web. The security concerns of site operators is part of the reason so few sites support uploadable code.
- Stand-alone applications. Today a user installing a stand-alone application (usually distributed on CD-ROMs or as Internet download) either chooses to trust the application completely or chooses not to install the application. Security conscious users decide whether an application is trustworthy enough to be installed and executed based on the reputation of its provider. Large companies are more likely to be trustworthy than individuals or small companies. Today, most applications are shipped in forms (e.g., Windows executables) that are not supported by most code safety systems. Efforts to convince program vendors to ship programs in a form that is more amenable to current code safety systems (e.g., source code or Java byte codes) are unlikely to be successful. Instead, we need code safety systems that can

efficiently and conveniently enforce policies on applications as they are commonly distributed.

- Constraining security-critical programs. A system administrator installing security-critical programs such as a remote login shell, an ftp server, a mail server, or a web server should be able to enforce specific constraints on their behavior. Although many of these programs do provide security configuration options (for example, a web server can be configured to allow access only to certain types of files), it would be useful to have an independent system that enforces these constraints as well as additional constraints. Using a separate code safety tool would have the advantage that the system administrator can use the same system to configure security constraints on different programs and to configure global constraints that apply to all programs. In addition, a code safety system independent of an application is not vulnerable to application bugs. There may be bugs in the code safety system, but if it is simple and extensively used, it is likely to have fewer security vulnerabilities than an application-specific mechanism.

In order to be useful in these contexts, Naccio implementations should securely enforce safety policies, and should be versatile enough to support a wide range of precise policies encompassing useful constraints on program behavior. Those policies should be defined in a way that makes them easy to define, understand and modify. It should be possible to produce Naccio implementations with a reasonable amount of effort. Finally, Naccio must be efficient enough so that even users without critical security needs will be willing to use it. These goals are often conflicting. Naccio seeks to expand the scope and precision of policies that can be enforced, as well as improve the policy-definition mechanisms, without substantially compromising security or efficiency and convenience.

1.3.1 Security

Security is an essential property of any code safety system. A secure code safety system correctly enforces the selected policy, even in the presence of motivated and knowledgeable attackers. Every system has vulnerabilities, but security systems should strive to eliminate known vulnerabilities and reduce the likelihood that attackers can find and exploit unknown vulnerabilities.

While proving a system is secure is generally infeasible for any non-trivial system, there are design approaches that are more likely to lead to secure systems. A simple design is more likely to be secure than a complex one, since flaws in a simple design are more likely to be detected and corrected. Further, it is more likely that a simple design can be implemented correctly than a complex design. A corollary to the simplicity goal is to have a small trusted computing base. If the security-critical part of the system can be isolated and kept small, it may be possible to verify its correctness or at least to carefully review the code.

1.3.2 Versatility

To be useful, a code safety system must be able to enforce useful policies. The ideal policy would prevent every behavior the user considers harmful but never issue a violation for behavior the user considers desirable. No such universal policy exists since it is impossible to perfectly distinguish harmful and desirable behavior. Indeed, behavior that is desirable for one program (such as `rm` deleting a file) would be considered harmful for other programs.

Supporting a wide range of policies means that policies can be defined to constrain many different program behaviors. For example, a system that does not provide any way to constrain thread creation cannot prevent denial-of-service attacks that create a huge number of threads.[4] A system that allows constraints on what files may be opened for reading or writing, but does not support any way of constraining what may be done with those files after they are opened must either prohibit writing entirely or allow attacks that fill up the local disk.

The other aspect of policy precision is the generality of the policy definition mechanisms. Some systems support policy checking based on setting a fixed set of parameters such as a list of readable and writeable files or an upper bound on network usage. This excludes a wide class of useful policies where the constraints are more dynamic or depend on other factors. For example, a useful policy might constrain what files may be written based on the command line or the history of user interactions; another policy might make the network usage bound a function of the number of keystrokes pressed by the user.

A completely general system would support policy checking using a universal programming language and with access to the entire state and history of the program execution. Such generality leads to complication in both policy definition and enforcement, and is probably not necessary for most practical policies. Instead, successful systems will make compromises based on providing sufficient generality to define most useful policies but enough limitations to make efficient and reliable enforcement feasible.

1.3.3 Ease of Use

A code safety system is useful only if it can enforce policies that place useful constraints on program behavior. In addition, there must be a way to define those policies. If it is too difficult or cumbersome to define policies, only predefined policies will be available to typical users. Only the most sophisticated experts will be able to create new policies, and obtaining a customized policy will be an expensive and time-consuming proposition.

Defining a policy requires good understanding of security requirements, but should not require extensive understanding of the execution platform. A policy definition mechanism that defines policies in terms of system calls on a particular platform can only be used by an elite group of platform experts. It is easy for even experts to forget about obscure system calls that can be used to manipulate resources leading to exploitable vulnerabilities. Naccio seeks to simplify policy definition by expressing policies in terms of manipulations of abstract resources that are not tied to a particular platform implementation, but correspond to things users understand like files and network connections.

1.3.4 Ease of Implementation

Since we hope that many implementations of Naccio will be developed, it is important that a Naccio implementation for a new platform can be produced with a reasonable amount of effort. Although some work will inevitably be required to support a new platform, Naccio's design should maximize reusability across platforms. It should also be clear what needs to be done to produce a Naccio implementation for a new platform, once the relevant properties of that platform are understood.

---

[4] One such attack that has been used to crash Windows 95/98 systems [Cnet99b].

1.3.5 Efficiency

The normal behavior of a code safety system is to do nothing noticeable to the user. A code safety system should be apparent only in the unusual situation where a program is about to violate the policy. This means the time and effort required to prepare a program to run with a selected policy enforced should be minimal. Most users will not even select the policy themselves, but rely on predefined policy settings established by their operating system or system administrator.

A code safety mechanism should also be transparent when a program runs, unless the policy is violated. It should not unduly affect the performance of the execution. The costs of enforcing a policy should be directly related to the complexity and ubiquity of the policy. It is reasonable that there be a significant overhead associated with enforcing a policy that monitors every byte written to files, but unreasonable for there to be any noticeable overhead for a policy that limits what directories can be read. Typical access control policies should be enforced with negligible overhead.

1.4 Contributions

This thesis presents a novel solution to the problem of constraining the behavior of program executions. We focus on addressing the limited class of policies supported by traditional code safety systems and the inadequate mechanisms they provide for defining policies.

Several other recent research projects have also attempted to expand the class of policies that a code safety system can enforce, most notably Ariel [Pandey98] and SASI [Erlingsson99]. Like Naccio, Ariel and SASI enforce policies by transforming programs. Naccio, Ariel and SASI can all enforce similar classes of policies. The key differences between Naccio and these and other projects are:

- Naccio is the first code safety system that defines safety policies in terms of abstract resource manipulations. This makes safety policies easier to write and understand, and means the same policy can be enforced on different platforms.
- Naccio is the first code safety system to use a two-stage process where policy compilation is separate from program transformation. This allows time-consuming optimizations that improve execution performance to be performed at policy compilation time, while allowing a policy to be enforced on an execution of a new program with low overhead.

Section 7.3 describes Ariel and SASI in more detail and clarifies the subtle differences in the classes of policies they can define.

Much was learned by building two Naccio prototype implementations and using them to define policies and enforce them on executions. Some specific contributions resulting from this experience include:

- We showed that it is possible to obtain the benefits of a large class of enforceable policies without sacrificing run-time performance when simple policies are enforced.
- We devised a specialization of dead code elimination that can be used to eliminate unnecessary checking code in code safety systems. This helps achieve our goal of only paying overhead for security checking when useful checking is being done.
- We gained an understanding of the tradeoffs involved in enforcing policies at different levels (for example, at the level of system calls or the level of machine instructions). The

Naccio architecture provides a clear framework for understanding what is lost or gained by selecting a particular level where policies are enforced.

- We clarified what properties must be guaranteed to ensure the integrity of wrapper-based checking mechanisms and designed mechanisms that provide these guarantees on the JavaVM and Win32 platforms.
- We introduced language features for creating groups of related resource operations. These groups can be used to define safety policies more easily and robustly.
- We introduced new mechanisms for combining safety properties based on intersection and weakening. These mechanisms are sufficiently powerful to enable easy expression of a wide class of policies, but simple enough to be readily understood and efficiently implemented.
- We developed a framework that can be reused to produce Naccio implementations for additional platforms with reduced effort.

Although the policy enforcement architecture is designed with the policy definition mechanisms in mind, they are separable. It would be reasonable to use different enforcement mechanisms to enforce policies defined using Naccio's definition mechanisms. Conversely, Naccio's enforcement architecture could be used to enforce policies defined in some other way.

1.5 Overview of Thesis

Chapter 2 introduces the Naccio architecture, describes its components and presents an example that shows how a policy is defined, compiled and enforced on a program execution. Chapter 3 describes how safety policies are defined. Chapter 4 describes how a platform is described in terms of its resource manipulations and how the platform interface can be altered to expand the class of policies that can be defined.

The next two chapters describe issues relating to enforcing policies in general as well as implementation issues involved in the two prototype implementations. Chapter 5 discusses what is done to compile a policy irrespective of the target application. Chapter 6 explains what is done to enforce a policy on a particular program execution.

Chapter 7 describes related work in code safety and program transformation. Chapter 8 evaluates Naccio's potential and examines vulnerabilities in the architecture generally, and in the prototype implementations specifically. Chapter 9 suggests future work and Chapter 10 summarizes the thesis and draws conclusions.

This Software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee warrants that it will not use or redistribute the Software for such purposes.

Sun JDK Noncommercial Use License

Chapter 2

Naccio Architecture[5]

Naccio is a system architecture for defining safety policies and enforcing those policies on executions. Conceptually, Naccio takes a program and a description of a safety policy, and produces a new program that behaves like the original program except that it is constrained by the safety policy. The Naccio architecture includes platform-independent languages for describing resources, general languages for specifying a safety policy in terms of constraints on those resources, and a family of platform-dependent languages for describing system calls in terms of how they manipulate resources. It also provides a framework for implementing policy enforcement mechanisms by transforming programs. This chapter provides an overview of the architecture. Chapters 3 and 4 describe how safety policies are defined. Chapters 5 and 6 describe issues involved in implementing the architecture and relate experience from building the two prototype implementations.

2.1 Overview

Suppose we wish to enforce a policy that limits the total number of bytes an execution may write to files. An implementation will need to maintain a state variable that keeps track of the total number of bytes written so far. Before every operation that writes to a file, we need to check that the limit will not be exceeded. One way to enforce such a property would be to rewrite the system libraries to maintain the necessary state and do the required checking. This would require access to the source code of the system libraries, and we would need to rewrite them each time we wanted to enforce a different policy. If the operating system were upgraded, the policy would need to be rewritten.

Instead, we could write wrapper functions that perform the necessary checks and then call the original system functions. To enforce the policy, we would modify target programs to call the wrapper functions instead of the protected system calls. Though wrappers are a reasonable implementation technique, they are not an appropriate way to describe safety policies since creating or understanding them requires intimate knowledge of the underlying system. To implement a policy that places a limit on the total number of bytes that may be written to files, one would need to identify and understand every system call that may write to a file. For even a

---

[5] Parts of this chapter are based on [Evans99].

supposedly simple platform like the Java API, this involves dozens of different routines. Changing the policy would require editing the wrappers, and there would be no way to use the same policy on other platforms.

Naccio's solution is to express safety policies at a more abstract level and to provide a tool that compiles these policies into the wrappers needed to enforce a policy on a particular platform. Safety policies are defined by associating checking code with abstract resource manipulations. A platform is characterized by how its system calls manipulate resources.

Figure 1 shows the Naccio system architecture. It is divided into a *policy compiler* and a *program transformer*. The policy compiler is run once per policy-platform pair. The policy compiler takes a definition of a resource use policy and a platform interface that describe an execution platform and produces a policy-enforcing platform library and a policy description file that encodes the transformations the program transformed must do to produce a program altered to enforce the policy. Since policy compilation is a relatively infrequent task, we trade off execution time of the policy compiler to make program transformation fast and to reduce the run-time overhead associated with safety checks. Once a policy has been compiled, the resulting policy-enforcing platform library and policy description file can be reused for each application on which we want to enforce the policy. Section 2.2 discusses the inputs and outputs of the policy compiler, and Chapter 5 provides details on how the policy compiler works.

The program transformer is run for each application-policy pair. It reads the policy description file produced by the policy compiler to determine what transformations need to be done to enforce the policy on an execution, and rewrites the program accordingly. The transformations typically include replacing calls to a platform library with calls to a policy-enforcing platform library produced by the policy compiler. In addition, the program transformer must ensure the necessary low-level code safety properties to prevent malicious programs from being able to tamper with the safety checking. Once the transformed program has been produced, it can be run normally and the policy will be enforced on the resulting execution. Section 2.3 discusses what the program transformer must do to enforce a policy, and Chapter 6 provides details on how this is done.

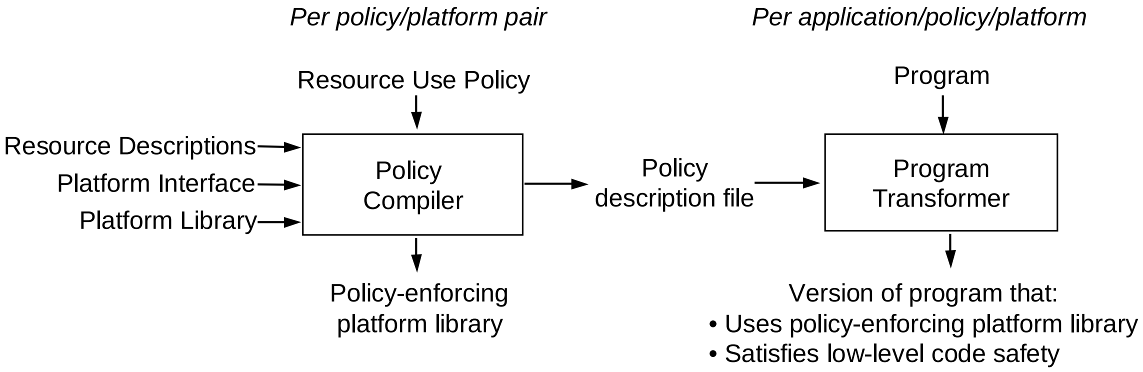


Figure 1. Naccio Architecture.

The left side of the figure depicts what a policy author does to generate a new policy. The right side shows what happens the first time a user elects to execute a given program enforcing that policy. The program transformer is run with an argument that identifies the policy description file to use.

An implementation of Naccio is characterized by the kind of program it transforms; the format and content of the platform libraries it uses; and the level of its platform interface, which determines the level at which it must transform the platform libraries and programs. We have built two Naccio prototype implementations: Naccio/JavaVM that enforces safety policies on JavaVM classes and Naccio/Win32 that enforces safety policies on Win32 executables. Although the design is intended to be general enough to apply to most modern platforms, the details and results in this thesis are derived from experience with these prototype implementations.

2.2 Policy Compiler

The policy compiler takes files describing a safety policy and an execution platform, and produces what is needed to enforce the policy. The input files consist of *resource descriptions* that provide a way to refer to resource manipulations abstractly; a *platform interface* that describes a particular execution platform in terms of those resource descriptions; a *platform library*, the unaltered code provided by the platform implementation (for example the Java API classes or Win32 system DLLs), and a *resource use policy* that specifies the constraints on program behavior to be enforced. For most policies, the resource descriptions and platform interface are treated as a fixed part of the implementation and the policy author writes a resource use policy.

A resource description defines a resource object and a list of *resource operations* that identify different ways of manipulating that resource object. For example, a resource description for a file system has a resource operation corresponding to writing bytes to a file. A resource use policy defines a safety policy by attaching checking code to these resource operations. Safety policies can be written and understood by looking solely at the resource descriptions and resource use policy. Naccio defines a standard set of resources that must be provided by any Naccio implementation. Policies defined in terms of those resources are portable and can be enforced without any extra effort on any platform for which a Naccio implementation is available. Policies defined in terms of the standard resources are known as *standard safety polices*. A challenge in designing Naccio is to choose a set of standard resource descriptions that can be used to define most typical safety policies, but that correspond precisely to the way actual resources are manipulated on different platforms. Chapter 3 describes how safety policies are defined, summarizes the contents of the standard resource library, and discusses the range of policies that may be expressed as standard safety policies.

A platform interface provides an operation specification of an execution platform in terms of a set of resource descriptions. The platform interface is a collection of wrappers that map concrete operations in a particular platform to the abstract resource manipulations described by the resource descriptions. The platform interface hides platform details from a policy author who need only look at the resource descriptions. A platform interface may be defined at different levels ranging from hardware traps to machine instructions to the system API to an application-specific library. For the most part, we focus on platform interfaces at the level of the system API since it is usually a well-defined interface and it provides a convenient place to interpose checking code. Platform interfaces at lower levels would be necessary to support policies that involve resource manipulations that are not visible through API calls. Platform interfaces at higher levels may be useful if we wish to support policies that apply to library or application level resources. If a policy author wishes to express a policy that cannot be defined in terms of the available resource descriptions, new resource operations can be defined by altering the platform interface. Chapter 4 describes the platform interface, and illustrates how the platform interface can be altered to define safety policies that cannot be expressed using the standard resource descriptions.

The policy compiler analyzes the resource use policy, resource descriptions and platform interface and produces a *policy-enforcing platform library*. If the platform interface is at the level of a system API, the policy compiler may also read and analyze the platform library object code, such as the Win32 API DLLs or the Java API classes. This is used to produce a new version of the platform library that includes checking code necessary to enforce the policy but otherwise behaves identically to the original platform library.

The policy-enforcing platform library makes calls to *resource implementations*, routines that correspond to the resource operations. The resource implementations do checking as directed by the resource use policy. The resource use policy defines checking code associated with resource operations. The policy compiler translates the code from the resource use policy and turns these resource operations into routines that can be called by the policy-enforcing platform library. Much of the work of the policy compiler is platform-independent. It parses the resource descriptions and resource use policy into intermediate languages and weaves the checking code into the appropriate resource operations. The resource operations are then implemented using a platform-specific back end that translates the intermediate language into executable code that performs the necessary checking.

The platform interface specifies how system calls need to be wrapped to call the appropriate resource operations. If run-time performance were not a concern, Naccio could generate the platform interface wrappers once and switch which resource implementations are used to enforce different policies. However, this would mean the overhead of going through a wrapper for a system call that manipulates constrainable resources would always be required regardless of whether or not the policy in effect constrains those resource manipulations. Instead, the policy compiler generates a new wrapped platform library for every policy. This means wrappers need only be generated for system calls that manipulate constrained resources. Generating a policy-specific version of the platform interface wrappers also allows for other optimizations to be performed, as described in Section 5.5.

The other output of the policy compiler is a *policy description file* that contains a compact representation of the transformations the program transformer must carry out to enforce the policy. The policy description file identifies the location of the policy-enforcing platform library so the application transformer can make the necessary changes. In addition, it may include rules to rename routines to call wrappers in place of system calls. This may be necessary in certain cases (such as wrapping native methods in Java) where the policy compiler cannot replace the routine in the policy-enforcing library. Other rules list resource operations that must be called at the beginning of execution (*initializers*) and resource operations must be called immediately before execution completes (*terminators*).

2.3 Program Transformer

The program transformer is run when a user elects to enforce a particular policy on an application for the first time. In a typical deployment, a web browser or application installer would run it transparently before a new program is executed based on a user's security settings.

The program transformer reads a policy description file and a target program and performs the directed transformations to produce a version of the program that is guaranteed to satisfy the safety policy. For each program and selected policy, we need to run the program transformer once. Afterwards, the resulting program can be executed normally. The type of program transformed depends on the particular Naccio implementation. It could be source code or object code, although implementations of Naccio that support object code are more likely to be useful since many vendors are unwilling to ship source code. The prototype implementations handle programs that are JavaVM classes and Win32 executables.

The program transformer makes two main changes to the program: it replaces the standard platform library with the policy-enforcing platform library produced by the policy compiler, and it modifies the program to ensure that the resulting program satisfies the low-level code safety properties necessary to prevent malicious programs from circumventing or altering the policy checking mechanisms. Both changes are platform-dependent, and as a result not much of the program transformer can be reused across different Naccio implementations. In addition, if the policy requires calls to initializers or terminators, the program transformer inserts these calls.

Switching the library is usually fairly simple on most modern platforms in which the platform library is linked dynamically. For Naccio/JavaVM it involves changing the CLASSPATH or replacing class names; for Naccio/Win32 it involves replacing file names in the import table. Guaranteeing the integrity of policy checks is more complicated. Naccio implementations must prevent programs from writing to storage or code used in safety checking or manipulating resources without going through the policy-enforcing platform library. Useful techniques for doing this include statically verifying that the necessary properties hold, performing low-level transformations on the application code to guarantee the necessary properties, and using platform interface wrappers so that the necessary properties are enforced by all policies. Section 6.2 discusses what must be protected and how this is done in Naccio implementations.

Figure 2 shows a sample wrapped system call sequence in a transformed program. Instead of calling the system call in the platform library directly, the transformed program calls the wrapped version of the system call in the policy-enforcing platform library that was produced by the policy compiler. This routine calls resource operations as directed by the platform interface. It may also need to do some bookkeeping to determine the correct arguments to pass to the resource operations. For the example, the wrapper for WriteFile must convert the file handle into an abstract resource object that identifies the corresponding file. The resource operations implement the checking specified by the resource use policy. If the policy would be violated by the system call, the resource implementation calls a Naccio library routine that reports the policy violation and gives the user the option to terminate or alter the execution. If not, the original system call in

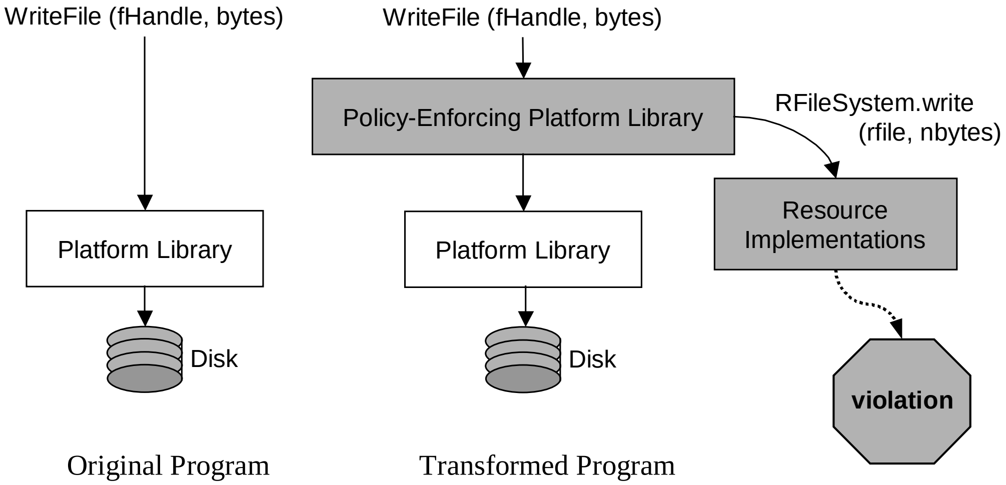


Figure 2. Wrapped system call sequence.

the platform library is called and the execution continues normally. Additional resource operations may be called after the system call returns. Depending on the Naccio implementation, the wrapper code may be embedded directly in the policy-enforcing platform library or kept as a separate library.

2.4 Walkthrough Example

This section walks through all the steps necessary to define and enforce a policy. It is not intended to be comprehensive, but to give the reader an idea of how all the pieces fit together. Chapters 3 through 6 describe each step in more detail. For this example, we consider using Naccio/JavaVM to enforce the LimitBytesWritten policy, which sets a limit of one million on the number of bytes that may be written to the file system on an execution of an application comprised of a set of Java class files. These steps would be substantially similar for Naccio/Win32 and implementations of Naccio for other platforms, but for simplicity this example is limited to Naccio/JavaVM.

This policy is expressed formally using Naccio's policy definition languages. We maintain a state variable that keeps track of the number of bytes written to the file system. We do this by declaring a new field named bytes_written that is associated with the RFileSystem resource object that represents the file system. This resource object is global over an execution, so the value of RFileSystem.bytes_written is maintained across the execution. This value needs to be incremented every time bytes are written to the file systems. The RFileSystem.postWrite resource operation corresponds to the point immediately after bytes were written to the file system, and we can maintain the value by attaching code that increments bytes_written to this resource operation. The bytes_written field declaration and updating code are encapsulated in a state block that can be reused by other safety policies.

To enforce the limit, we need to check that the limit will not be exceeded before allowing a write to proceed. We do this by attaching checking code to the RFileSystem.preWrite resource operation that corresponds to the point immediately before bytes will be written to the file system. This checking code compares the sum of the number of bytes already written (as recorded in the RFileSystem.bytes_written state variable) and the number of bytes about to be written to the limit enforced by the policy. If the limit would be exceeded, it issues a violation and gives the user an opportunity to terminate the execution. The code used to define this policy is shown in Figure 6 in Section 3.2.

The policy must be compiled before it can be enforced on an application execution. To compile a policy, we need an operation specification of the execution platform known as a platform interface. The platform interface describes concrete events in terms of the abstract resource descriptions used to define the policy. Naccio/JavaVM uses a platform interface at the level of the Java API (the java. classes). The Java API platform interface describes each method in the Java API by calling resource operations at the execution points defined by the resource descriptions. For example, the description of the RFileSystem.preWrite operation documents that it should be called before every write to the file system with a parameter that gives an upper bound on the number of bytes about to be written. The platform interface wrapper for the java.io.FileOutputStream.write(byte[]) method indicates that RFileSystem.preWrite should be called before the write method is called, and RFileSystem.postWrite should be called after the write method returns. The policy compiler produces a new version of the java.io.FileOutputStream class that replaces the write method with a wrapper that calls the resource operations as described by the platform interface around the original method. The

Naccio/JavaVM platform interface wrapper for the java.io.FileOutputStream class is shown in Figure 11 and discussed in Section 4.2.

The policy compiler also generates implementations corresponding to the abstract resource operations that are called by the generated wrapper classes. Naccio/JavaVM implements each resource using a Java class with a method that corresponds to each resource operation. Code from the resource use policy is woven into the resource implementations and translated to Java code. Section 5.3 explains how the policy compiler generates a resource implementation class.

A policy author or system administrator runs the policy compiler, and its output can be used to enforce the policy on any JavaVM program. The generated wrapper classes and resource implementations are stored in a protected directory and the policy compiler generates a policy description file that encodes the transformations needed to enforce the policy on an execution. When a user elects to enforce the policy on a program execution, the application classes are transformed according to the rules in the policy description file. For Naccio/JavaVM, this can involve simply setting the CLASSPATH so that the generated wrapper classes are found before the standard Java API classes. After this has been done, the application can be executed normally with the safety policy enforced on its execution. Chapter 6 describes the program transformer.

Figure 3 shows what happens at run-time to enforce the LimitBytesWritten policy on an application that creates a java.io.FileOutputStream and writes an array of bytes to it. The original FileOutputStream class is replaced with a policy-enforcing wrapper version of the class, shown in the figure as lbw.FileOutputStream. The constructor for this class constructs an RFile object that is an abstract resource corresponding to the file associated with this output stream. This object is stored in an instance variable of the lbw.FileOutputStream object, and will be passed to resource operations like RFileSystem.preWrite. After constructing this object, the original constructor executes normally and stores the RFile object in a new instance variable. Unlike the RFile object, the RFileSystem is a global resource so there is only one RFileSystem object for the entire execution. When the execution calls java.io.FileOutputStream.write(byte[]), the wrapper for this method will call the resource operation RFileSystem.preWrite, passing in the RFile object associated with this FileOutputStream and the size of the array. The RFileSystem.preWrite implementation contains the checking code from the policy, and will issue a violation if the policy would be violated by the write method call. Otherwise, it returns and the original write method is executed. After it completes, RFileSystem.postWrite is called. This method contains the code that increments bytes_written.

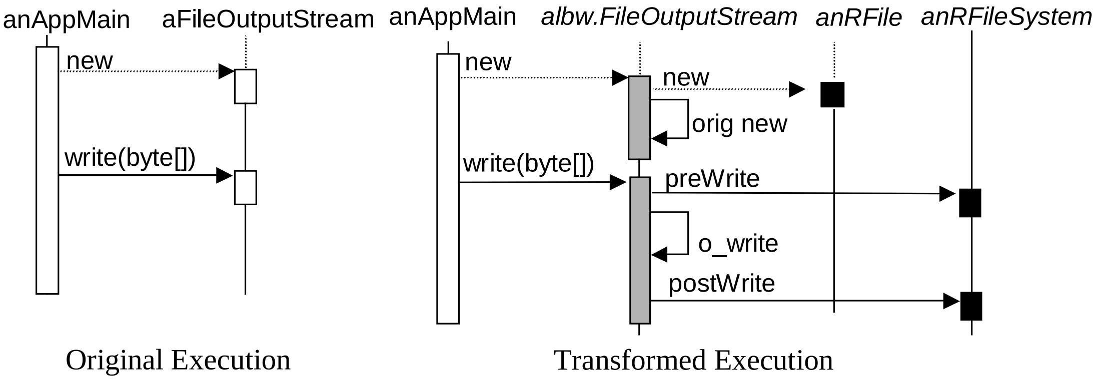


Figure 3. Interaction diagram for enforcing LimitBytesWritten.

For an explanation of the interaction diagram notation see [Gamma95]. The gray objects are classes modified by Naccio. The black objects are classes generated by Naccio.

Chapter 3

Defining Safety Policies

This chapter describes how Naccio is used to define safety policies. For standard policies, we consider the resource descriptions and platform interface to be a fixed part of the system and express a policy only in terms of resource use constraints. Standard polices are portable across Naccio implementation platforms. The standard resources are chosen so that many useful safety policies can be defined as standard safety policies. This includes policies that place access constraints on system resources such as reading and writing files and opening network connections, and policies that place limits on consumption such as the number of files that may be touched or the number of bytes that may be written to the file system. This chapter discusses resource descriptions, specifying safety policies that constraint resource manipulations, the contents of the standard resource library and the limits on expressiveness for standard safety policies. In the next chapter, we discuss how a platform interface is used to specify a platform in terms of how it manipulates resources and consider policies that can be expressed by changing the platform interface.

3.1 Resource Descriptions

A program runs by executing a sequence of instructions. Those instructions modify the state of the processor and may affect devices attached to the machine such as its hard drive, network connection and display. We can view everything a program can manipulate as a *resource*. A safety policy imposes constraints on how a program manipulates resources. In order to define a safety policy, we need a precise way of referring to resource manipulations.

Resource descriptions provide a way to identify resources and describe ways they are manipulated. Examples of resources include files, network connections, threads and displays; examples of manipulations are writing ten bytes to a file, opening a network connection to port 80 on `naccio.lcs.mit.edu`, increasing the priority of a thread, or opening a window. Resource descriptions are written in a platform-independent language, but they may describe platform-specific resources such as the Windows registry. Naccio includes a set of standard resource descriptions that encompass the resource manipulations that are common on nearly all platforms and are relevant for many security policies.

We describe resources by listing their operations. Typical resource descriptions have no state or implementation. They are merely hooks for use in defining safety policies. Resource descriptions may use primitive types including `int`, `float` and immutable `Strings`. These types are defined by Naccio to have the expected semantics. The meaning of a resource operation is indicated by informal documentation. This documentation should be clear and precise to the policy author, but is not sufficiently formal to be processed by a machine.

Policy authors read resource descriptions, but do not need to modify them for typical policies. A policy is expressed by associating checking code with resource operations. The essential promise is that a transformed program will invoke the related resource operation with the correct arguments whenever a particular event occurs. It is up to the policy compiler and platform interface to ensure that this is the case.

Figure 4 shows two resource descriptions related to the file system. It declares the `RFileSystem` resource object that represents to the file system as a whole, and the `RFile` resource object that identifies a single file or directory. The `RFileSystem` resource has operations that correspond to manipulating files and directories. The `RFile` resource only contains a constructor for creating a resource object that identifies a particular file. The `global` modifier indicates that only one `RFileSystem` instance exists for an execution[6]. Resources declared without a `global` modifier are associated with a particular run-time object. Most of the `RFileSystem` operations take an `RFile` parameter to identify a particular file. Dividing a resource into a global resource for the actual manipulations and instance resources for identifying resources is a common paradigm. This division makes it easy to write policies that constrain system-wide resource use (for example, the total number of files that are opened), but provides an abstract way to identify specific objects such as files.

3.1.1 Resource Operations

The body of a resource description is a list of operations and groups. Each operation corresponds to a particular way of manipulating a resource. For example, the `openRead` operation corresponds to opening a particular file for reading. Its documentation prescribes that `openRead` is called before a file is opened for reading. It takes a parameter of type `RFile` that represents the file being opened.

The documentation associated with each resource operation must be precise enough so that policy authors can write policies that behave as expected. However, it should not be over specified in ways that prevent it from being applicable on different platforms. For example, what it means to open a file is a platform-specific notion. The essence of the open operations is given by the documentation for the read and write operations that indicate the relevant open operation must be called first. Platform-specific documentation may be necessary in some cases to clarify what resource operations mean. Given reasonable choices, however, policies can be reused across platforms with their intended meaning.

Resource manipulations may be split into more than one resource operation. For example, reading is split into the `preRead` and `postRead` operations. This division allows more precise safety policies to be expressed. Pre-operations allow necessary safety checks to be performed before the action takes place, while post-operations can be used to maintain state and perform additional checks after the action has been completed and more information is available. For this example, the actual number of bytes read may not be known until after the system call that does the read has completed.

---

[6] For now, we consider an execution to be all activity within a process, so that all applets running within a Java virtual machine are treated as part of the same execution by global resources. Section 9.3 discusses how deployments might define the scope of a resource differently.

```
global resource RFileSystem
 operations
   initialize ()                             Called when execution starts.
   terminate ()                              Called immediately before execution ends.

   openRead (file: RFile)                    Called before file is opened for reading.
   openAppend (file: RFile)                  Called before file is opened for appending.
   openCreate (file: RFile)                  Called before file is created for writing.  At this point in the
                                                execution, file must not exist..
   openOverwrite (file: RFile)               Called before file is opened for writing.  At this point in the
                                                execution, file exists.
   close (file: RFile)                       Called before file is closed.

   preDelete (file: RFile)                   Called before file is deleted.
   postDelete (file: RFile)                  Called after file is deleted.

   renameNew (file: RFile, newfile: RFile)   Called before file is renamed to new file newfile.  At this
                                                point in the exection, newfile must not exist.
   renameReplace (file: RFile, newfile: RFile)  Called before file is renamed to existing file newfile.

   makeDirectory (file: RFile)               Called before creating new directory file.

   preWrite (file: RFile, n: int)            Called before up to n bytes are written to file; file must have
                                                previously been passed to openCreate, openOverwrite
                                                or openAppend.
   postWrite (file: RFile, n: int)           Called after exactly n bytes were written to file.

   preRead (file: RFile, n: int)             Called before up to n bytes are read from file; file must
                                                have previously been passed to openRead.
   postRead (file: RFile, n: int)            Called after exactly n bytes were read from file.

   observeExists (file: RFile)               Called before revealing if file exists.
   observeWriteable (file: RFile)            Called before revealing if file is writeable.
   observeCreationTime (file: RFile)         Called before revealing creation time of file.
   observeList (file: RFile)                 Called before revealing files in directory file.
   … // other similar observe<X> operations elided

   setCreationTime (file: RFile)             Called before changing creation time of file.
   … // other similar set<X> operations elided

   group modifyExistingFile (file: RFile)    Called before contents of any existing file are modified.
      openOverwrite, openAppend, preDelete,
      renameNew (file: RFile, newfile: RFile): modifyExistingFile (file),
      renameReplace (file: RFile, newfile: RFile): modifyExistingFile (file),
      renameReplace (file: RFile, newfile: RFile): modifyExistingFile (newfile);

   group modifyFile (file: RFile)            Called before any file is altered or created.
      modifyExistingFile, openCreate,
      renameNew (file: RFile, newfile: RFile): modifyFile (newfile);

   group observeProperty (file: RFile)       Called before any property of file is revealed.
      observeExists, observeWriteable, observeCreationTime, …;

   … // Other groups elided.

global resource RFile
   operations
      RFile (pathname: String)               Constructs object corresponding to pathname.  Pathname
                                                is a canonical string that identifies a file.
```

Figure 4. File System Resources.

The RFile resource has only one operation, a constructor. It takes a string parameter that identifies a file in some platform-dependent way. The RFile resource objects have no state or operations provided to obtain information about what actual file a particular RFile object represents. Policies can add the necessary state and operations to determine properties of an RFile. Section 3.2.1 illustrates how this is done.

Two special resource operations are not associated with resource manipulations but represent the beginning and ending of executions. The initialize operation is called at the beginning of execution, before any program-directed file manipulation is done (file manipulations done by system initialization code may occur before initialize is called). The terminate operation is called after all program-directed file manipulations have completed. Most global resources provide initialize and terminate operations. They provide useful places to attach checking code or to initialize state associated with checking.

3.1.2 Resource Groups

Resource operations may also be grouped to make it easier to write safety policies. A resource group is a set of resource operations and other resource groups that correspond to similar manipulations. Grouping operations makes it easier to define policies that do not depend on specific manipulations. For example, the observeProperty group encompasses all resource operations that correspond to observing properties of a file. It includes the observeExists operation that is called before revealing if the given file exists and several other operations associated with observing properties of a file. Since some policies need to distinguish between observing whether a file exists and observing the size of a file, the RFileSystem resource description should have separate operations corresponding to each manipulation. Since many policies do not need to distinguish between the different ways of observing file properties, it is also useful to define a group that encompasses all the file observation operations.

A resource group is defined by listing the operations and groups it contains. All members in a resource group must map to the parameters of the group. The mapping is given by a function-call like syntax that calls the group name. Conceptually, the resource operation calls the group in the way given by the function call. For example, in the modifyExistingFile group list we use

```
renameNew (file: RFile, newfile: RFile) : modifyExistingFile (file),
```

to map rename, which takes two parameters, into the modifyExistingFile group, which takes a single RFile parameter. Since the only existing file modified by renameNew is the file corresponding to its first parameter, the group mapping passes this parameter to modifyExistingFile. For renameReplace, both the file and newfile already exist so two existing files are modified by the corresponding resource manipulation. The group definition for modifyExistingFile lists renameReplace twice with different mappings corresponding to each file modification.

If the group parameters and the member parameters match exactly, listing the operation name assumes the implicit mapping where the parameters correspond directly. For example, the observeExists resource operation and observeFile resource group both take one parameter of type RFile, so listing observeExists is sufficient.

3.2 Safety Properties

A safety property attaches checking code to resource operations or groups. The simplest safety property specifies that a particular resource manipulation is not permitted. For example,

```
property NoDeleting {
   check RFileSystem.preDelete (file: RFile) {
      violation (“File deletion prohibited.”);
   }
}
```

defines a property that issues a violation before an application would delete a file. The documentation given in the RFileSystem resource description shown in Figure 4 indicates that the preDelete operation is called before a file is deleted. The body of the check clause calls the violation function provided by the Naccio library. It will display a dialog box containing the text of the violation and information on the safety property that is about to be violated. The user is presented with the option to terminate the execution, or to ignore the violation and allow execution to continue.

As it is defined, the NoDeleting property is probably not satisfactory. It prevents explicit deletion of existing files, but does not prevent deleting a file by overwriting its contents or renaming another file to its name. A more comprehensive property that prevents any modification of existing files could be defined as:

```
property NoBashingFiles {
   check   RFileSystem.openOverwrite (file: RFile),
           RFileSystem.openAppend (file: RFile),
           RFileSystem.preDelete (file: RFile),
           RFileSystem.renameNew (file: RFile, newfile: RFile),
           RFileSystem.renameReplace (file: RFile, newfile: RFile) {
      violation (“Destructive file manipulation prohibited.”);
   }
}
```

A simpler definition would use the modifyExistingFile group that groups all resource operations that alter the contents of existing files:

```
property NoBashingFiles {
   check RFileSystem.modifyExistingFile (file: RFile) {
      violation (“Destructive file manipulation prohibited.”)
   }
}
```

Using resource groups makes the property more concise and easier to understand. It also means the property will not need to be changed if new resource operations are added as long as the modifyExistingFile group is appropriately amended.

3.2.1 Adding State

One problem with these properties is that the violation text provides no useful information about what file is being manipulated. The user cannot tell the difference between an execution that is about to alter a junk file and one that is about to alter an important file. As is, it is impossible to do this by modifying only the check action since the RFile object passed to the resource operations does not contain any information about the file it corresponds to.

In order to track this information, state must be added to the `RFile` resource. Naccio supports this using a state block:

```
stateblock FileNames augments RFile {
   addfield name: String;

   precode RFile (pathname: String) {
      name = pathname;
   }

   helper getName () returns String {
      return name;
   }
}
```

This augments the `RFile` object with `name`, a `String` field representing the name of the file. The `precode` block associated with the `RFile` constructor sets `name` to the value of its parameter, a `String` that canonically identifies a particular file. This constructor is called to create an `RFile` object before any operation that requires it is called. Since all `RFile` objects are created using this constructor, the name is available wherever an `RFile` object is used. Safety properties can refer to the name of an `RFile` object `rfile`, using `rfile.name` or by calling the helper method `getName`. It is useful to keep the state maintenance and safety property checking code separate, since many safety properties use the same state.

Figure 5 shows the `NoBashingFiles` property modified to use the file name information to produce a more helpful violation message. The `requires` clause identifies the state block that defined `RFile.getName`. The state block is defined in a separate file that is found using a naming convention. Properties can include multiple state blocks as long as multiple state blocks do not use the same field or helper routine name.

```
property NoBashingFiles {
   requires FileNames;
   check RFileSystem.modifyExistingFile (file: RFile) {
      violation (“Destructive manipulation of file:” + file.getName ());
   }
}
```

Figure 5. NoBashingFiles property.

3.2.2 Use Limits

State can be also be used to make policies more precise. For example, a property based on `NoBashingFiles` could do a test on the file name to allow modification of files in the `/tmp/` directory but prohibit all other modifications of existing files. State can also be used to define policies that place limits on the amount of a resource that may be used over the course of an execution. For example, the `LimitBytesWritten` property shown in Figure 6 places a limit on the total number of bytes that may be written to the file system.

To enforce a limit on the number of bytes that may be written, the property must keep track of the total number of bytes written. The `TrackBytesWritten` state block does this by adding a field to the `RFileSystem` resource and defining a `postcode` action for the `write` operation. The body of the `postcode` action will happen after all checking code associated with the resource operation. Hence, when `bytes_written` is used in the check action of `LimitBytesWritten`, its value is the total number of bytes written already not including the upcoming call. After all the checking code has executed, the value is updated to account for the upcoming write.

```
stateblock TrackBytesWritten augments RFileSystem {
    addfield bytes_written: int = 0;
    postcode postWrite (file: RFile, n: int) {
        bytes_written += n;
    }
}

property LimitBytesWritten (limit: int) {
    requires TrackBytesWritten, FileNames;
    check RFileSystem.preWrite (file: RFile, n: int) {
        if (bytes_written + n > limit)
            violation ("Attempt to write more than " + limit + " bytes. Already written " +
                bytes_written + " bytes, writing up to " + n + " more to " + file.getName () + ".");
    }
}
```

Figure 6. LimitBytesWritten Safety Property.

3.2.3 Composing Properties

This simplest way to combine properties is to intersect them using the & operator. The intersection of two safety policies allows an execution only if both policies allow the execution. That is to say, the intersection of one or more safety properties issues a violation whenever any of the individual properties would issue a violation. If more than one of the properties would issue a violation for the same resource operation, the violation reported by the first property appears first. Intersecting safety properties is equivalent to merging all the check clauses into one property in the same order they were intersected.

Another way to combine two safety properties is to weaken a property with permissions that override violations. All the previous properties have been expressed negatively, in terms of issuing violations before a prohibited manipulation is about to happen and implicitly allowing everything else. An alternative way of defining properties is to assume nothing is allowed unless it is explicitly permitted. This has the advantage that is it less likely for a policy author to accidentally allow something dangerous. Conversely, it is more likely that a policy author will forget to allow something that is needed by a harmless program. To avoid arguments about which approach is preferable, Naccio supports both and provides rich enough property combination mechanisms to allow both positive and negative properties to be used.

A permission uses allow to indicate that the given resource manipulation is permitted. For example,

```
permission AllowModifyDir (path: String) {
    requires FileNames;
    check RFileSystem.modifyExistingFile (file: RFile) {
        if (NCheck.inDirectory (file.getName (), path)) allow ();
    }
}
```

allows files in the directory identified by path to be modified. The inDirectory library function does a comparison to determine if the file is contained within the directory identified by path. A property cannot use both allow and violation.

By default, Naccio policies assume everything is allowed. Hence, a permission only makes sense when it is combined with a negative property. The universal negative policy would associate a check clause with every resource operation that simply issues a violation (this is what the

DisallowAll policy used in Section 8.4 does). Defining policies in terms of permissions that override the universal negative policy would satisfy the principle of fail safety that recommends disallowing all security-relevant behavior that is not explicitly allowed [Saltzer75]. This approach makes sense when there is a small, fixed set of security-relevant behavior, but becomes cumbersome when the class of behavior considered to be security-relevant is large and flexible. It would be undesirable if all policies had to be rewritten when a new resource is added. Since this is expected to be fairly common with Naccio, Naccio's default is to allow everything that is not explicitly prohibited.

Hence, permissions are only useful in a context where some manipulations are already prohibited. When a property is weakened by a permission, violations in the property are overridden by allowances in the permission. For example,

```
property NoBashingExceptTmp {
   (NoBashingFiles weaken AllowModifyDir ("/tmp/")) weaken AllowModifyDir ("/u/evs/tmp/")
}
```

defines a property that issues a violation whenever a file not in the /tmp/ or /u/evs/tmp/ directories is modified. The allowances in a positive property override violations in a negative property. If the weakening property calls allow on a particular invocation of a resource operation, no violations will be issued from that resource operation. Another way to express the same property would be to compose the positive properties first:

```
property NoBashingExceptTmp {
   NoBashingFiles weaken (AllowModifyDir ("/tmp/") & AllowModifyDir ("/u/evs/tmp/"))
}
```

Weakening is useful for combining new policies with standard policies that describe commonly allowed behavior. For example, JDKFilePermissions is a standard set of permissions that allow files loaded by the JDK initializations and AWT to be read. A new safety policy that prevents reading files except those loaded by the JDK initializations can be expressed easily by writing a no reading property that disallows all file reading and weakening it with JDKFilePermissions.

In order to enforce a policy on an execution, all parameters must be bound to real values. This is done by instantiating all parameterized properties with parameters. We call a property in which all parameters are bound a *resource use policy*. All parameters must be manifest constants. Figure 7 shows the LimitWrite resource use policy that disallows modification of any existing file or writing more than one million bytes to the file system. Properties that have no parameters can also be used directly as resource use policies.

```
policy LimitWrite {
   NoBashingFiles & LimitBytesWritten (1000000)
}
```

Figure 7. LimitWrite resource use policy.

3.3 Standard Resource Library

The standard resource library is a set of resource descriptions that correspond to the security-relevant resource manipulations that are common to most modern platforms. The standard resource library does not attempt to exhaustively cover all possible ways of manipulating resources, but instead is designed to include the manipulations commonly used in security policies that are universal enough to apply to most platforms. Since all Naccio implementations provide the same standard resource library, policies written in terms of these resources are portable across different platforms.

The standard resource library includes the RFile and RFileSystem resources introduced in Section 3.1, as well as resources corresponding to the network, the display, system threads, audio devices, and the system environment. It contains a total of 122 resource operations in thirteen resource descriptions. Additional resources may be needed as new devices are attached to the system. For example, if a camera is used a corresponding resource should provide operations that correspond to taking and transmitting pictures. There may also be resources that are unique to a particular platform. For example, Naccio/Win32 includes a resource representing the Windows registry.

Network

In most modern operating systems, the network can be used in three distinct ways: a persistent connection can be created to a remote host, and data sent and received through it; a server socket can be created to listen for incoming connections; and individual datagram packets may be sent or received without a persistent connection. Since policies should be able to distinguish between each type of network use, we provide different resource objects for identifying them. Conversely, the network operations should make it easy to write network use policies that place restrictions on the remote hosts that may be contacted and limits on the number of bytes transmitted. To support easy definition of both kinds of policies, the network resources provide operations corresponding to the different types of network connections, but also groups operations so policies that do not depend on the type of network connection can be defined concisely.

The network resources are shown in Figure 8. Unlike the file system resources, the RNetConnection resource maintains some state and provides an observer. An *observer* is a routine that reveals some information about a resource but does not modify anything. The RNetConnection stores the local and remote addresses of the connection in state variables when an RNetConnection is constructed. The observers make these values available through a function call.

The observers can be used in resource group member lists to map members to the group operation. This is done in the definition of the connectRemoteAddress resource group that takes an RNetAddress parameter representing the remote address. To make the preOpenConnection resource operation match the parameter types of the connectionRemoteAddress group, we need to convert its RNetConnection parameter into the appropriate RNetAddress object corresponding to the remote address. We do this by calling the getRemoteAddress observer defined by the RNetConnection resource.

Display

The display is represented by the RDisplay global resource, and RWindow resource objects identify individual windows. The main security threats involving the display are denial of service annoyance attacks that take over the screen with superfluous windows. A more serious threat is attacks that create rogue windows that appear to be part of a legitimate application and trick the user into providing trusted information (such as a password) to a malicious program. This threat can be mitigated by a policy that requires that all windows created from untrusted programs have a distinctive appearance that distinguishes them from trustworthy windows.

The RDisplay resource includes operations for creating new windows and for setting properties of windows or manipulating existing windows. It also contains operations that correspond to enabling a window to receive events from the mouse or keyboard and receiving those events. These could instead be treated as separate resources, but since events are usually directed at a window it is convenient to include them with the display. By using a state block to track user input events, policies can determine if a resource manipulation is permitted based on the history of user activity. Since windowing systems are likely to vary more across platforms than other

```
global resource RNetwork
   operations
      initialize ()                                Called at the beginning of an execution.
      terminate ()                                 Called immediately before execution terminates.

      preOpenConnection (connection: RNetConnection)    Called before opening connection.
      postOpenConnection (connection: RNetConnection)   Called after opening connection.
      closeConnection (connection: RNetConnection)      Called after closing connection.

      preOpenListener (listener: RNetListener)     Called before opening listener for server connections.
      postOpenListener (listener: RNetListener)    Called after opening listener for server connections.
      preAccept (listener: RNetListener)           Called before accepting a connection using listener.
      postAccept (listener: RNetListener, connection: RNetConnection)
                                                   Called after accepting connection using listener.
      closeListener (listener: RNetListener)       Called after closing listener.

      openDatagramPort (port: RNetListener)        Called before opening port for datagrams.
      closeDatagramPort (port: RNetListener)       Called before closing port.

      preSendDatagram (local: RNetAddress, remote: RNetAddress, nbytes: int)
         Called before up to nbytes are sent from local to remote using a datagram.
      preSendConnection (connection: RNetConnection, nbytes: int)
         Called before up to nbytes are sent through connection.

      preReceiveDatagram (local: RNetAddress, nbytes: int)
         Called before a datagram may be received at local.
      postReceiveDatagram (local: RNetAddress, remote: RNetAddress, nbytes: int)
         Called after nbytes are received from remote to local.

      ... // other operations for postSend, preReceive and postReceive for datagrams and connections elided

      group connectRemoteAddress (address: RNetAddress)     Called before any contact with address.
         preOpenConnection (connection: RNetConnection)
            : connectRemoteAddress (connection.getRemoteAddress ()),
         postAccept (listener: RNetListener, connection: RNetConnection)
            : connectRemoteAddress (connection.getRemoteAddress ()),
         preSendDatagram (local: RNetAddress, remote: RNetAddress, nbytes: int)
            : connectRemoteAddress (remote),
         postReceiveDatagram (local: RNetAddress, remote: RNetAddress, nbytes: int)
            : connectRemoteAddress (remote);  // can't know remote before receive, must check after

      group preSend (remote: RNetAddress, nbytes: int)
         preSendDatagram (local: RNetAddress, remote: RNetAddress, nbytes: int)
            : preSend (remote, nbytes),
         preSendConnection (connection: RNetConnection, nbytes: int)
            : preSend (connection.getRemoteAddress (), nbytes);

      … // similar groups for postSend, preReceive and postReceive elided
      … // operations related to multicasting and revealing hostnames elided.

resource RNetConnection
   state local, remote: RNetAddress; // Identfy the local and remote addresses for this connection.
   operations
      RNetConnection (l: RNetAddress, r: RNetAddress)
         Constructs an RNetConnection object for communication between l and r.
         { local = l; remote = r; }
   observers
      getLocalAddress () returns RNetAddress       { return local; }
      getRemoteAddress () returns RNetAddress   { return remote; }

// RNetAddress and RNetListener not shown.
```

Figure 8. Network Resources.

resources, it is likely that Naccio implementations will add additional operations to the RDisplay resource to include platform-specific operations that provide more precise ways of constraining display use.

Threads

The RSystemThreads global resource provides operations corresponding to manipulating threads. It includes operations for creating new threads or thread groups, starting and destroying threads, suspending and resuming threads, changing the priority of a thread, and revealing information about a thread or thread group. The RThread and RThreadGroup resources are used to identify threads and groups of related threads.

Audio

The speaker can be used in an annoyance attack. To support policies that constrain its use, the RAudio global resource contains operations corresponding to ringing the system bell and playing audio files.

System Environment

The RSystem resource is used to collect operations that do not correspond well to a conceptual resource. It includes operations for observing and setting environment variables, and is often extended with platform-specific system operations.

The RSystem resource also includes special initialize and terminate operations that are called at the beginning of an execution. The RSystem initializer is called before any other global resource initializers. The RSystem terminator is called after every other global resource terminator. The RSystem initializer is also unique in that it has an argument that passes in the command-line arguments. A policy can use a state block that attaches checking code to RSystem.initialize to record these values, and then use the value of the command-line arguments to determine if a resource manipulation is permitted.

3.4 Policy Expressiveness

In standard safety policies, the effects of checking code are limited to raising violations, modifying internal state, and doing computations that are invisible to the user. The policy has no noticeable effect on an execution (other than a performance penalty) unless a violation is detected. We can view a Naccio standard safety policy as a predicate on an execution – it is true if no violation is issued, and false if a violation is issued.

Schneider defines Class EM, a class of enforcement mechanisms that work by monitoring a target system and terminating any execution that is about to violate the policy [Schneider98]. Security kernels, reference monitors, and nearly all run-time based enforcement mechanisms are in Class EM. The set of policies that can be enforced by mechanisms in Class EM is defined as those policies that can be expressed as predicates on execution prefixes.

A *security policy* is defined as a predicate on a set of executions. A program satisfies a security policy if the predicate is satisfied by the set of all possible executions it can produce. Policies like information flow require knowledge of more than one execution, since it is not clear whether a particular execution of a program reveals information without knowing what other executions do. Hence, these policies cannot be enforced by mechanisms in Class EM. Enforcing these policies requires static analysis of the program text.

Those security policies that can be defined as a predicate on a single execution are known as *security properties*. Not all security properties, however, are in Class EM, since they may depend on knowing the future. For example, liveness properties depend on knowing something must happen at some future point in an execution. Class EM mechanisms cannot enforce liveness properties since they can only probe what has already happened.

The subset of security properties that can be defined by looking only at the past and present are defined to be *safety properties*. A safety property is a predicate on an execution prefix. If it is false at some point in an execution, it is false for all following execution points.

The policies that can be enforced using an enforcement mechanism in Class EM are a subset of safety properties. The subset is defined by how much information the enforcement mechanism can probe. An enforcement mechanism that can probe all system information after every instruction could enforce all safety properties.

To satisfy the requirements of class EM, the probe should have no effect on the system and should be completely unnoticeable by the executing program. This is not possible if the probe is implemented in software running on the same machine as the program it is probing. At a minimum, it uses CPU cycles that would otherwise be available to the execution. In some cases, it may need to manipulate resource also. For example, to enforce the `NoBashingFiles` property introduced in Section 3.2 using Naccio/JavaVM, it may be necessary to examine the file system to determine if a file already exists (Section 4.2.2 shows how the platform interface is written to do this). We consider resource manipulations done by the checking code to be separate from the behavior of the program. These manipulations are done without any checking enforced. This means policy authors must be wary that an attacker cannot exploit code introduced to do checking.

Aside from the side effects introduced by probing, Naccio standard safety policies are in Class EM. They observe the behavior of an execution through resource operations and issue a violation to terminate execution when a policy violation is about to occur. The subset of safety properties that can be defined as Naccio standard safety properties is defined by the resource operations defined by the standard resource library. Naccio can detect violations and observe and modify state only at execution points corresponding to resource operations, and can only observe system information available through parameters to resource operations (as well as some global system information that can be observed through calls to Naccio library functions).

Certain safety properties cannot be defined using the standard resources. For example, since `RFileSystem.preWrite` takes an integer parameter revealing the number of bytes to be written but does not have a parameter corresponding to the actual data written, we cannot write a policy that constrains the actual values of bytes that may be written. In the next chapter, we describe how resource operations are given meaning using a platform interface and how new resource operations and safety policies can be defined by altering the platform interface. In addition, by removing some of the restrictions placed on standard safety policies, Naccio can be used to define and enforce policies that alter program behavior. Because these policies do not simply probe system information and decide to terminate an execution, they do not fit Schneider’s definition of a security policy. As a result, Naccio is not strictly in Class EM.

Chapter 4

Describing Platforms

The previous chapter showed how a safety policy is defined in terms of resource descriptions. To have meaning, there must be a way of viewing the way a particular platform manipulates actual resources in terms of those abstract resource descriptions. This is done using a platform interface, an operational specification of a platform in terms of its resource manipulations. Naccio implementations include a platform interface that describes the platform in terms of the standard resource library. Changing the platform interface allows new resource operations to be defined and more safety policies to be described and enforced. We call policies that are defined by altering the platform interface *extended safety policies*.

4.1 Platform Interfaces

In order to enforce a policy defined in terms of abstract resources, we need a way to model an execution in terms of those resources. The platform interface provides an operational specification of a concrete execution platform in terms of a set of resource descriptions. A different platform interface is needed for each execution platform and each set of resource descriptions. The platform interface provides a way to map events during a program execution to abstract resource manipulations. Since the specification is operational, it is easy for the policy compiler to convert it to code that calls the resource operations in the appropriate way.

We can view the platform interface as a probe that can see certain system events. Based on those events, it can execute bookkeeping code and call abstract resource operations that perform the checking necessary to enforce a policy. A Naccio implementation determines what events are visible to the probe, and where in the execution chain it sees them. The events visible determine what resource operations can be defined and this limits the class of policies that can be expressed and enforced. For example, if the platform interface can only see manipulations of the file system then resource operations relating to manipulating the network cannot be defined. If the platform interface can see the entire state of the machine before and after every instruction, then all policies in class EM can be enforced. Policies defined using a platform interface that can see all system events, however, are likely to be cumbersome and expensive to enforce. Instead, the platform interface is defined at a level that allows only certain events to be seen. For example, a platform interface might be defined in terms of calls in the system API. This would make the platform interface easier to create and understand, and would simplify the work of the policy compiler and program transformer. It would not support the definition or enforcement of policies that constrain resources that can be manipulated without going through system API calls, such as referencing a memory location.

A Naccio implementation must also determine where in the execution chain the platform interface probe is done. This level determines the trust boundary between what is described by the platform interface and what is considered part of the program. Operations below the level described by the platform interface execute without safety checking and are assumed to manipulate resources in the way specified by the platform interface. Operations above the level described by the platform interface are transformed to perform the safety checking defined by the resource use policy. The lowest conceivable place for the platform interface is at the level of physical hardware devices. For example, a disk drive controller could be designed to call a resource operation before writing a bit to the disk or a firewall could monitor network traffic and call appropriate resource operations. This would require hardware support not readily available today. If it were available, however, this would allow safety policies to be enforced without trusting anything other than the hardware controllers. The difficulty would be mapping these events to resource operations. The disk controller can provide information about which segment on the disk is being written, but probably cannot convert that to a meaningful pathname. This requires operating system support, and would expand the trusted computing base to include the relevant system code. Another difficulty with a hardware-level platform interface is the problem of associating a particular manipulation with the program that caused it. Again, the hardware traps will need to rely on operating system level code to map requested actions to the program instigating them and the appropriate safety policy. This would require substantial run-time overhead. Since the effective policy is not known until the application is determined, the overhead is required even for simple policies or unconstrained executions. For most situations, hardware-level safety checking is not practical or appropriate. There are situations, however, where safety is crucial enough that it is desirable to place the safety checking at as low a level as possible so that bugs in the system library do not lead to policy violations. For example, it would be appropriate for medical devices (such as the Therac-25 mentioned in Section 1.1) with custom hardware and control software.

The next level to consider for the platform interface is at the level of machine instructions. A platform interface at this level would allow any instruction to be mapped to resource operations. Trust would be confined to the behavior of individual machine instructions, although as with the hardware-level checking, it is likely that some information provided by the operating system would be necessary in mapping instructions to meaningful objects. The main problem with defining a platform interface at the level of machine instructions is that it would be hard to produce and understand. Recognizing all sequences of instructions that represent a function call, and defining a platform interface in terms of those instruction sequences is likely to be a cumbersome and error-prone task.

Above the individual machine instructions, we can consider a platform interface at the level of the system API. Typical modern operating systems have a protected kernel, and allow programs to manipulate most resources only through calls to routines provided by that kernel. The system API provides a convenient place for the platform interface since it is usually well documented and structured to provide an abstract way to manipulate resources. Placing the platform interface at this level has other advantages in implementing the policy enforcement mechanisms. Unlike lower-level platform interfaces that would require correspondingly low-level transformations of both the program code and system API code to enforce a policy, a policy defined at the level of the system API can be enforced by interposing checking code at system call boundaries. This requires that the execution platform provides a clear distinction between the system API and application code, and that this interface be maintained securely. One disadvantage of placing the platform interface at this level are that certain resource manipulations, such as allocating or referencing memory, may not be visible through calls to the system API. Another problem is that we must trust to system API implementation to manipulate resources in the way described by the platform interface. This makes the system API part of the trusted computing base and means attackers can exploit bugs in the system API to circumvent the safety policy. The other issue with a platform interface at the level of the system API is that it is necessary to ensure that programs cannot manipulate constrained resources without using the standard system API. Despite these disadvantages, the system API seems to be the best place for the platform interface for most Naccio implementations. Both of our prototype implementations use platform interfaces at the level of a system API. Naccio/JavaVM uses a platform interface that describes the Java API (classes in the `java.` packages) and Naccio/Win32 uses a platform interface at the level of the Win32 API. Sections 4.2 and 4.3 describe these platform interfaces.

We can also consider platform interfaces at a higher level. A platform interface could describe a commonly used library such as Microsoft Foundation Classes (MFC) that is implemented using the Win32 API. This would support more higher-level distinctions (and hence, more precise policies) than could be written with a platform interface at a lower level. For example, we could use a platform interface at the level of MFC to define different resource operations corresponding to opening a file selected by the user using a standard dialog box and opening a file without user prompting. Providing a similar distinction at a lower level would be possible, but very awkward. It would be necessary to examine the properties of the window to see if it looks like a standard file request dialog and the input from the user to determine what file was selected. Another option would be to write a platform interface that describes application level events. This would allow policies to be defined in terms of objects that are meaningful at the application level but not at the system such as application data structures. The problem with higher-level platform interfaces is that they only work for a subset of programs that use those higher-level libraries. Programs that manipulate constrained resources in other ways must be disallowed. This could be done by a static analysis that the code never uses system API calls directly. It would summarily reject many harmless programs, however, simply because they were not written using the higher-level library.

4.2 Java API Platform Interface

Naccio/JavaVM enforces safety policies on executions of Java programs that are collections of JavaVM classes. To enforce Naccio policies on Java classes, we need a platform interface that maps a Java execution to a sequence of abstract resource operations.

4.2.1 Platform Interface Level

Naccio/JavaVM uses a platform interface at the level of the Java API. Another reasonable option would be to put the platform interface at the level of individual byte code instructions. This would allow for resources to be described that correspond to manipulations done below the level of the Java API, such as memory references. All high-level system resources including the file system, network, and display are accessible to Java programs only through native methods. If an untrusted program is not permitted to install its own native methods or call native methods installed by other programs, the only way it can manipulate these resources is through calls to the Java API. Placing the platform interface at the level of the Java API allows nearly all security-relevant manipulations to be constrained and allows the platform interface to be described at a well-documented and well-defined level. Further, a platform interface at the level of the Java API provides a convenient place to introduce wrappers.

To define the platform interface, we could examine the API specification and write a wrapper for each API routine that describes its resource usage. This would involve substantial work, and depend on the API specification being correct and describing resource usage of all routines in sufficient detail. We can simplify the task of writing a Java API platform interface, however, by noting that all relevant resource manipulations must eventually be done by native methods. This means a platform interface for the Java API could describe the resource manipulations done by native methods explicitly, and determine the resource manipulations done by other routines based on their code (either statically or at run-time). This would limit the amount of work necessary to write the platform interface to describing the native methods in a particular Java API library implementation.[7]

A problem with this approach is that it ties the platform interface closely to a particular API implementation, instead of to the specification of the Java API. Since we must describe private native methods, the same platform interface could not be reused with a different implementation of the Java API. The other problem with specifying the platform interface at the level of native methods is that it may be difficult to determine enough information about the context of a call to pass appropriate information to the resource operations.

Instead, the Naccio/JavaVM platform interface describes only the specified parts of the Java API. It does not describe any private API methods since the Java API does not specify these. It does, however, support a pass-through semantics so that not every API routine needs to be described explicitly. For routines that are not explicitly described, the routines they call are checked as if they were called directly by the untrusted program. We can use implicit specifications only for routines that do not directly or indirectly call any native methods whose behavior is not explicitly described. Hence, the platform interface must explicitly describe any API routine that has a native implementation, that calls a private native method directly, or that calls a private native method indirectly through calls to other routines that are not explicitly specified (these routines must be private, otherwise they would have be explicitly specified). Other API routines may be described implicitly by passing checking through to the routines they call. This limits the size of the platform interface since most routines can be described implicitly. It does, unfortunately, tie our platform interface to a particular implementation of the API. It should be easy to adapt it to a different implementation, however. All that is required is to write wrappers for any routines that are specified implicitly in the old implementation but implemented using native methods or indirect calls to unspecified native methods in the new implementation. This is preferable to requiring that the platform interface explicitly describe every routine of the Java API.

The code body of member wrappers is written in a simple Java-like language. This code may call resource operations, call Naccio library routines, use and set wrapper state, and do computation using that state, parameters, and local variables. It may use if-else statements to control flow, but not while or for loops. When a wrapper calls a resource operation, the necessary safety checking is performed. If the policy would be violated, the user has the opportunity to terminate execution. The hash token (#) marks the execution point where the original routine is called. Hence, resource operations that correspond to events that occur before the described resource manipulation must be called before the hash mark and resource operations that correspond to events that occur after the described resource manipulation must be called after the hash mark. The return value of the call to the original routine is stored in a local variable named `result` and may be used in the remainder of the wrapper body. For example, the wrapper for `java.io.File.delete` is defined by:

```
wrapper boolean delete () {
   RFileSystem.preDelete (rfile);
   #;
   if (result) { RFileSystem.postDelete (rfile); }
}
```

It calls `preDelete` before the delete method executes. The `rfile` argument is an instance variable of type `RFile` introduced by the platform interface. If the checking code associated with `preDelete`

---

[7] In fact, Sun's implementation of the JDK 1.1.6 API uses 567 native methods.

issues a violation and the user chooses to terminate the execution, the actual delete method is never executed. Otherwise, the delete method is executed and its boolean return value is identified by the local variable result. If the call returned true (meaning the deletion completed successfully), the postDelete operation is called. If this completes without issuing a violation, the result is returned and execution continues normally after the call.

4.2.2 File Classes

Figure 9 shows the platform interface wrapper for the java.io.File class. For each visible routine defined by java.io.File, the class wrapper either provides a wrapper that describes the behavior of the member in terms of its effects on abstract resources, or declares the member to be a passwrapper. The resource use of the passwrapper routines is accounted for implicitly by the routines their implementation calls. Checking is done for these routines as though they were called from the application directly.

The java.io.File wrapper adds a state variable, rfile, of type RFile that will be associated with each java.io.File object. This state is used to map a java.io.File object to a resource object that identifies the corresponding actual file. It is up to the member wrappers to maintain this state. Hence, each constructor initializes it to an RFile object. Instead of constructing a new object directly, RFile objects are maintained using the RFileMap helper class (shown in Figure 10). This ensures that the same RFile object is used for all manipulations on the same concrete file even if there are multiple java.io.File or java.io.FileDescriptor objects that refer to that file. Storing the rfile state is not strictly necessary, since the wrappers could use the file map to obtain the appropriate RFile object every time it is needed. Keeping the rfile in an instance variable, however, is likely to have better performance that repeatedly looking it up in the file map.

Routines that are implemented without calling native methods are declared as passwrappers. This avoids the need to understand the behavior of these members in detail, but means the platform interface is tied to a particular Java API implementation (in this case, Sun’s JDK 1.1.6). If another implementation used a native method to implement getAbsolutePath or called an unwrapped native method in its implementation, the platform interface would need to be modified to explicitly describe how it manipulates resources. When Naccio/JavaVM processes a platform interface, it issues warnings if a passwrapper member relies on an unwrapped native method (either by calling it directly, or by calling unwrapped non-native methods that indirectly call an unwrapped native method). Since all visible native methods must have wrappers, this is only possible if the implementation of a passwrapper member calls a private native method directly or through calls to other unwrapped routines.

The declaration of the java.io.File wrapper uses requiredif clauses. These clauses are not necessary for correctness but are used by the policy compiler to eliminate unnecessary wrappers to reduce run-time checking overhead. The clause requiredif RFile, RFileSystem in the declaration of the java.io.File wrapper indicates that the wrapper is only necessary if either the RFile or RFileSystem resources have meaningful checking. Without this clause, the policy compiler would not be able to determine this automatically and would generate a policy-enforcing library that requires more run-time overhead than should be necessary. Section 5.2 describes how the policy compiler analyzes the platform interface in conjunction with the resource use policy to determine which wrappers are necessary.

```
wrapper java.io.File
            requiredif RFile, RFileSystem {
    requires RFileMap;
    state RFile rfile;

    wrapper File (String path) {
        #; rfile = RFileMap.lookupAdd (this);
    }

    wrapper File (String path, String name) {
        #; rfile = RFileMap.lookupAdd (this);
    }

    wrapper File (java.io.File dir, String name) {
        #; rfile = RFileMap.lookupAdd (this);
    }

    passwrapper String getAbsolutePath();
    passwrapper String getCanonicalPath();
    passwrapper String getParent();

    wrapper boolean exists () {
        RFileSystem.observeExists (rfile); #;
    }

    wrapper boolean canWrite () {
        RFileSystem.observeWriteable (rfile); #;
    }

    wrapper boolean canRead () {
        RFileSystem.observeReadable (rfile); #;
    }

    wrapper boolean isFile () {
        RFileSystem.observeIsFile (rfile); #;
    }

    wrapper boolean isDirectory () {
        RFileSystem.observeIsFile (rfile); #;
    }

    wrapper long lastModified () {
        RFileSystem.observeLastModifiedTime (rfile);
        #;
    }

    wrapper long length () {
        RFileSystem.observeLength (rfile); #;
    }

    wrapper boolean mkdir () {
        RFileSystem.makeDirectory (rfile); #;
    }

    passwrapper boolean mkdirs ();

    wrapper boolean renameTo (java.io.File dest) {
        if (dest.exists ())
            RFileSystem.renameReplace
                (rfile, dest.rfile);
        else
            RFileSystem.renameNew
                (rfile, dest.rfile);
        #;
    }

    wrapper String[] list() {
        RFileSystem.observeList (rfile); #;
    }

    passwrapper String[]
        list (java.io.FilenameFilter filter);

    wrapper boolean delete () {
        RFileSystem.preDelete (rfile); #;
        if (result) RFileSystem.postDelete (rfile);
    }
}
```

Figure 9. Platform interface wrapper for java.io.File class.

```
helper class RFileMap { // Mapping between java.io.File and java.io.FileDescriptor objects and RFile
    static private Hashtable fmap = new Hashtable ();

    public static RFile add (java.io.File f) {
        RFile rf = new RFile (path);
        fmap.put (f.getAbsolutePath (), rf);
        return rf;
    }

    public static void addReference (java.io.FileDescriptor d, RFile f) { fmap.put (d, f); }
    public static RFile lookup (Object f) { return (RFile) fmap.get (f); }

    public static RFile lookupAdd (Object f) {
        RFile rf = lookup (f);
        if (rf == null)
            if (f instanceof java.io.File) rf = add ((java.io.File) f);
            else if (f instanceof java.io.FileDescriptor)
                … // Treat file descriptors specially (standard streams are null).
        return rf;
    }
}
```

Figure 10. RFileMap helper class.

```
wrapper java.io.FileOutputStream requiredif RFile, RFileSystem {
    requires java.io.RFileMap;
    state RFile rfile;

    helper void doOpen (java.io.File file) {
        rfile = RFileMap.lookupAdd (file);
        if (file.exists ()) RFileSystem.openOverwrite (rfile);
        else RFileSystem.openCreate (rfile);
    }

    wrapper FileOutputStream (java.io.File file)   { doOpen (file); #; }
    wrapper FileOutputStream (String file)          { doOpen (new java.io.File (file)); #; }

    wrapper FileOutputStream (java.io.FileDescriptor file) {
       rfile = RFileMap.lookup (file);
        if (rfile != null) RFileSystem.openOverwrite (rfile); // File must already exist since its a descriptor
       #;
    }

    wrapper FileOutputStream (String file, boolean append) {
        File tmp = new File (file);
        if (append) {
            rflile = RFileMap.lookupAdd (tmp);
            RFileSystem.openAppend (rfile);
        } else
            doOpen (tmp);
         #;
    }

    wrapper void write (int b) {
        // Although Java int's are four bytes, write only writes the low order byte.
        if (rfile != null) RFileSystem.preWrite (rfile, 1);
        #;
        if (rfile != null) RFileSystem.postWrite (rfile, 1);
    }

    wrapper void write (byte data[]) {
        if (rfile != null) RFileSystem.preWrite (rfile, data.length);
        #;
        if (rfile != null) RFileSystem.postWrite (rfile, data.length);
    }

    wrapper void write (byte b[], int off, int len) {
        if (rfile != null) RFileSystem.preWrite (rfile, len);
        #;
        if (rfile != null) RFileSystem.postWrite (rfile, len);
    }

    wrapper void close () {
        if (rfile != null) RFileSystem.close (rfile);  #;
    }

    wrapper java.io.FileDescriptor getFD () {
        #; RFileMap.addReference (result, rfile);
    }
}
```

Figure 11. Platform Interface wrapper for java.io.FileOutputStream class.

Other Java API classes that manipulate files have wrappers that describe their behavior in terms of the RFileSystem resource. One example is the java.io.FileOutputStream class, shown in Figure 11. As with java.io.File, the wrapper for java.io.FileOutputStream maintains an RFile object representing the actual file corresponding to this output stream. This state can be null, if the FileOutputStream does not correspond to a file (for example, if it is the standard output stream).

Because the RFileSystem resource provides different resource operations for overwriting an existing file and creating a new file, the FileOutputStream constructors must distinguish between opening existing and new files. This is done by the doOpen helper method. It calls java.io.File.exists to determine whether to call the openOverwrite or openCreate resource operation. Internal routine calls in platform interface wrappers always call the unwrapped versions of routines. Hence, the call to exists bypasses the wrapper and does no safety checking.

4.2.3 Network Classes

The platform interface for the network classes is less straightforward than it was for the file classes, since the network resource is manipulated in several different ways and socket transmissions are done using generic input and output stream classes.

```
wrapper java.net.Socket {
   requires NRegulatedNetworkInputStream, NRegulatedNetworkOutputStream, SocketHelp;
   state RNetConnection rnc;

   wrapper Socket (String host, int port) {
      rnc = new RNetConnection (new RNetAddress (SocketHelp.getLocalAddress (),
                                new RNetAddress (SocketHelp.absoluteName (host), port));
      #;
      rnc.getLocalAddress ().setPort (getLocalPort ()); // Local port is not known until after constructor.
      RNetwork.postOpenConnection (rnc);
   }

   … // Other constructors similar.

   wrapper InputStream getInputStream()
      // Only necessary if preReceive or postReceive does checking.
      requiredif RNetwork.preReceive (RNetAddress, RNetAddress, int),
                 RNetwork.preReceive (RNetConnection, int),
                 RNetwork.postReceive (RNetAddress, RNetAddress, int),
                 RNetwork.postReceive (RNetConnection, int) {
      #;
      result = new NCheckedNetworkInputStream (result, rnc);
   }

   wrapper OutputStream getOutputStream ()
      requiredif RNetwork.preSend (RNetAddress, RNetAddress, int),
                 RNetwork.preSend (RNetConnection, int),
                 RNetwork.postSend (RNetAddress, RNetAddress, int),
                 RNetwork.postSend (RNetConnection, int) {
      #;
      result = new NCheckedNetworkOutputStream (result, rnc);
   }

   … // other methods elided
}
```

Figure 12. Platform interface for java.net.Socket.

```
helper class NCheckedNetworkOutputStream extends java.io.FilterOutputStream {
   RNetConnection rnc;

   public NCheckedNetworkOutputStream (OutputStream os, RNetConnection r) {
      super (os);
      rnc = r;
   }

   public void write (int b) throws IOException {
      RNetwork.preSend (rnc,1);
      super.write (b);
      RNetwork.postSend (rnc, 1);
   }

   // Other write methods overrided similarly.
}
```

Figure 13. NCheckedNetworkOutputStream helper class.

Figure 12 shows the java.net.Socket platform interface. The getInputStream and getOutputStream methods return stream objects used for sending and receiving data through a socket. Since the RNetwork resource provides operations corresponding to sending or receiving bits over the network, the platform interface must ensure that the appropriate resource operations are invoked when these streams are used. The wrappers for the get stream methods accomplish this by returning a subclass of InputStream or OutputStream constructed using the result of the original method and the RNetConnection object. These subclasses call resource operations when data is received or sent through a network connection. Figure 13 shows excerpts from the NCheckedNetworkOutputStream helper class; NCheckedNetworkInputStream is similar.

In addition to the persistent stream used by java.net.Socket, the network may be manipulated by sending or receiving datagram packets and by using server sockets that listen for incoming connections. The platform interfaces for java.net.DatagramSocket and java.net.ServerSocket describe the Java API classes corresponding to these manipulations. Other classes such as java.net.URLConnection also provide routines that can be used to manipulate network connections, and are described appropriately by the platform interface.

4.2.4 Extended Safety Policies

This section demonstrates how a policy that cannot be defined using the standard resources can be defined by using an altered platform interface. First, we introduce a standard policy that places a limit on the rate of network usage. This policy is then improved by modifying the platform interface.

A safety property that limits the total amount of data sent or received over the network can be written similarly to the LimitBytesWritten property introduced in Figure 6. Instead of tracking bytes written to files, this policy would track bytes sent over the network using the RNetwork preSend and postSend operations. Such a policy would be useful in detecting obviously bad behavior from programs that are permitted to use the network but not expected to send or receive a large amount of data. A more generally useful policy would allow for a limit to be placed on the rate of network usage instead of the total amount. Writing such a policy depends on dividing time into quanta and keeping track of the number of bytes sent during the current time quantum. Figure 14 shows a policy that limits the rate of network transmissions by delaying sending. It prevents the application from sending more than maxBytes bytes over the network in an ms millisecond time period.[8]

Although this policy constrains network bandwidth as desired, it is far from satisfactory. If the preSend operation is called with a higher number of bytes than maxBytes, it leads to a violation since there is no way to alter the send to conform to the rate. Further, if the number of bytes doesn't exceed the quantum limit but is slightly higher than the number allowed in the remaining time quantum, it stalls until the current time quantum completes instead of sending part of the transmission right away. Without changing the platform interface, there is no way to fix these problems since the resource operation has no control over the system call it is constraining. By modifying the platform interface, however, and integrating it with the policy information, we can change the way network transmissions are done to improve the policy.

```
stateblock TrackSendRate (timeQuantum: int) augments RNetwork {
   addfield bytesSent: int = 0;
   addfield timeStart: int;

   helper updateTimer () {
      if (naccio.library.Time.getCurrentTime () - timeStart > timeQuantum) {
         // The current time quantum is finished, reset. Ignores numeric wrap around.
         bytesSent = 0; timeStart = naccio.library.Time.getCurrentTime ();
      }
   }

   helper waitForQuantum () {
      if (naccio.library.Time.getCurrentTime () - timeStart < timeQuantum) {
         naccio.library.Time.sleep (timeQuantum - (naccio.library.Time.getCurrentTime () - timeStart));
      }
      updateTimer ();
      assert (bytesSent == 0); // check a new quantum was started
   }

   precode postSend (connection: RNetConnection, nbytes: int) {
      updateTimer (); bytesSent += nbytes;
   }
}

property NetLimitSendRate (maxBytes: int, ms: int) {
   // Send up to maxBytes in time ms
   requires TrackSendRate (ms);
   precheck RNetwork.preSend (connection: RNetConnection, nbytes: int) {
      updateTimer ();
      if (bytesSent + nbytes > maxBytes) {
         if (nbytes <= maxBytes) waitForQuantum ();
         else
            violation ("Network send rate exceeded.  Maximum of " + maxBytes + " bytes per " + ms
                       + "ms. Already sent " + bytesSent + " this quantum; attempting to send "
                       +  nbytes + " bytes.");
      }
   }
}
```

Figure 14. Policy that limits network send rate by delaying transmissions.

---

[8] To be more precise, since all the sending in one checking quantum could occur at the end, and all the sending in the next occurs at the beginning, it is possible that there is some quantum-length time slice in which nearly 2 * maxBytes are transmitted. More generally, for n adjacent time slices, the total number of bytes sent is not greater than (n + 1) * maxBytes.

Figure 15 shows a policy that splits and delays network sends to conform to a requested bandwidth usage.[9] The SoftSendLimit property includes an alterinterface clause that modifies the platform interface using the alternate wrapper for java.net.Socket shown in Figure 16 (a similar wrapper for java.net.DatagramSocket is not shown). It replaces the wrapper for getOutputStream to construct and return an NRegulatedOutputStream object instead of the NCheckedNetworkOutputStream returned by the standard wrapper.

Excerpts from the definition of NRegulatedOutputStream are shown in Figure 17. It loops until the entire array of bytes is transmitted. Each iteration calls RNetwork.quantumSendAvailable (defined by the SoftSendCounter state block) to find out how much bandwidth is remaining in the current time quantum. Since quantumSendAvailable is defined to stall until the end of the time quantum if no more bandwidth use is allowed, it always returns a positive value. It then calls the RNetwork.preSend resource operation for the actual send, calls write to send the data, and then calls the RNetwork.postSend resource operation.

```
stateblock SoftSendCounter (sendLimit: int, timeQuantum: int) augments RNetwork {
   requires TrackSendRate (timeQuantum);

   helper quantumSendAvailable () returns int { // Number of bytes more that can be sent this quantum
      updateTimer ();
      if (bytesSent >= sendLimit) waitForQuantum ();
      return (sendLimit - bytesSent);
   }
}

property SoftSendLimit (limit: int, tq: int) {
   requires SoftSendCounter (limit, tq);
   alterinterface java.net.Socket: RegulatedSendSocket,
                  java.net.DatagramSocket: RegulatedSendDatagramSocket;

   precode RNetwork.preSend (connection: RNetConnection, nbytes: int) {
      // No checking necessary, but use assertion to make sure platform interface is doing the right thing.
      assert (nbytes + bytesSent <= sendLimit);
   }
}
```

Figure 15. Policy that limits bandwidth by splitting up and delaying network sends.

```
alter wrapper java.net.Socket {
   requires java.net.NRegulatedOutputStream;

   replace wrapper OutputStream getOutputStream () {
      #;
      result = new NRegulatedOutputStream (result, rnc);
   }
}
```

Figure 16. RegulatedSendSocket wrapper modification code.

---

[9] We assume the application does not depend on how sends are packaged. This is not necessarily true, and some applications will fail if network sends are split.

```
helper class NRegulatedOutputStream extends java.io.FilterOutputStream {
   RNetConnection rnc;

   public void write (byte b[]) throws IOException {
      long offset = 0;

      do {
         long avail = RNetwork.quantumSendAvailable ();
         if (avail + offset > b.length) avail = b.length - offset; // Can send the rest

         // Assumes no other threads send since call to quantumSendAvailable.
         RNetwork.preSend (rnc, avail);
         out.write (b, offset, avail);
         RNetwork.postSend (rnc, avail);

         offset += avail;
      } while (offset < b.length);
   }

   … // Other methods elided.
}
```

Figure 17. NRegulatedOutputStream helper class (excerpted).

For simplicity, this implementation assumes there are no other program threads that may send data over the network between the call to quantumSendAvailable and the call to postSend. If this were to happen, two threads could attempt to use the same available bandwidth leading to a failure of the assertion defined in the check body for RNetwork.preSend in the SoftSendLimit property. To prevent this, an implementation could use a semaphore to lock the RNetwork resource when quantumSendAvailable is called and release it after calling postSend. While locked, future calls to quantumSendAvailable would stall until the lock is released.

In addition to altering existing wrappers, policy authors can replace wrappers completely, remove existing wrappers, or add new wrappers. This can be done to provide fine control over behavior in ways that is not possible in checking code itself. It can also be done to define new resource operations that can be used like standard resource operations in defining safety policies.

4.3 Win32 Platform Interface[10]

Naccio/Win32 is intended to provide code safety on a variety of Windows operating system platforms. The Win32 API is used by many Windows-based operating systems including Windows 95, Windows 98, Windows NT, Windows CE and Windows 2000. Although Naccio/Win32 is intended to support many Win32-based operating systems, this discussion focuses on Windows NT (which is believed to be the basis for all future Windows operating systems including Windows 2000).

Like most modern operating systems, Windows NT has a protected kernel that provides system calls that can be used to manipulate the hardware and control basic operating system functions. Application processes are confined to their own virtual address space but can make calls to kernel

---

[10] For more details on the Win32 platform interface, see [Twyman99]. This section is largely based on that document.

code by executing a trap instruction. On top of the kernel, NT provides several OS environments including Win32, Posix and MS-DOS. Each OS environment is implemented by a protected subsystem – a user-level process that receives requests from client process using Local Procedure Call (LPC) messages. All OS environments in Windows NT are implemented using the Win32 subsystem, which makes direct calls to the kernel. Since programming using LPC messages would be tedious, the Win32 subsystem provides an application program interface (API) that allows programs to access the Win32 subsystem using function calls. All Windows implementations implement this API using a dynamic link library (DLL). DLLs are linked when a program is loaded or during execution, but are not statically linked into an executable.

4.3.1 Platform Interface Level

There are several options for the level of the Naccio/Win32 platform interface. The lowest level would be at the level of machine instructions for a particular machine architecture, such as Intel x86. This would mean alternate machine architectures (such as the DEC Alpha) could not be supported without writing a new platform interface. Further, enforcing policies at that level would require modifying the NT kernel and involve substantial complexity. Since using a different version of the kernel for programs that enforce different policies is not readily possible within the Windows architecture, it would be necessary to integrate the checking hooks into the standard kernel and determine at run-time which policy should be enforced. As a result, most of the overhead of the most expensive safety policy needs to be incurred for even trusted programs running with no policy constraints.

The next possible level for the platform interface is the NT kernel. The platform interface could describe calls provided by the NT kernel and enforce policies by interposing checking code around calls to the kernel. This would require modifying the protected subsystems. Since Windows 95/98 does not use protected subsystems, one disadvantage of this approach is that it would only work for Windows NT. Another problem with trying to write a platform interface at the level of the NT kernel is that there is no definitive documentation available for the kernel calls, and platform interface authors would need to rely on guesswork to describe their behavior correctly.

The most appropriate choice is to put the platform interface at the level of the Win32 API. This is the lowest level that is standardized and well documented. It is shared across Windows operating systems and machine architectures. Selecting a platform interface at the level of the Win32 API restricts the target programs to those written to the Win32 API, so programs written for the Win16, MS-DOS or Posix subsystems are not supported. Importantly, though, it means that a single platform interface can be used across all Win32 systems, and as a result, much of the policy compiler and program transformer can be reused across all Win32 systems. Placing the platform interface at the level of the Win32 API offers several advantages in the ease of creating an implementation and its efficiency at both transformation and execution time. The Win32 API is encapsulated entirely in DLLs. This provides a clear interface where the platform interface wrappers can be interposed. A disadvantage of placing the platform interface at this level is that Naccio/Win32 must ensure that programs cannot circumvent safety checking by manipulating resources without using the Win32 API, for example, by making direct kernel calls. Section 6.2.2 discusses what must be done to provide the necessary assurances.

Placing the platform interface at a higher level would be likely to exclude too many programs. One option would be writing a platform interface for the Microsoft Foundation Classes (MFC). Many Win32 programs are written using MFC, a C++ library that provides object-oriented abstractions of the Win32 API. A platform interface at this level would support policies that could take advantage of information that is readily available in MFC calls but harder to extract from Win32 API calls. For example, it could treat opening a file selected by the user from a standard dialog box differently from normal file opening. If the only platform interface available describes MFC, it would be necessary to prevent the application from making direct calls to the Win32 API. Another problem is that MFC may be linked either statically or dynamically. If it is linked statically, the MFC calls cannot easily be securely detected and replaced with wrappers. On the other hand, combining a Win32 API platform interface with an MFC platform interface would be a viable option. This would allow polices to be enforced on programs that call the Win32 API directly, but allow more permissive policies to allow additional resource manipulations from programs that use MFC. The Naccio prototypes do not support multiple level platform interfaces, although it is a clear extension of the architecture. Section 9.2 discusses extensions to Naccio that would be necessary to support this.

4.3.2 Prototype Platform Interface

Several compromises were made to make creating the platform interface for the Naccio/Win32 prototype manageable. Because of the size and complexity of the Win32 API, the Win32 platform interface only describes a small subset of the API, focusing on the simple file manipulation calls. Hence, only policies defined using only the RFile and RFileSystem resources can be enforced.

The other major compromise taken to make Naccio/Win32 manageable is to express the platform interface using stylized C code that can be compiled directly using the macro definitions generated for the resource implementations. This eliminates the need for the policy compiler to parse and analyze the platform interface. It also removes the possibility to optimize out unnecessary wrappers, and means that the overhead required for simple policies is substantially more than would be the case if some simple optimizations were done. This was viewed as acceptable considering the proof-of-concept nature of the Naccio/Win32 prototype.

Figure 18 shows an excerpt from the Naccio/Win32 platform interface for the DeleteFileA system call. Since the platform interface is designed to be C code that can be compiled directly, it uses a naming convention to invoke resource operations. A resource operation is called by the resource type name followed by an underscore and the resource operation name. The policy compiler will define macros corresponding to these names that do the actual resource invocation. The wrapper calls RFileMap_addRFileByName to obtain an RFile object corresponding to the pathname. Since Win32 programs are not garbage collected, we use reference counting to manage object memory. When the returned RFile is no longer needed, the wrapper code calls RFile_release to indicate that the object reference is no longer needed.

```
BOOL wrapper__DeleteFileA (LPCTSTR pathname) {
   BOOL result;
   RFile rf = RFileMap_addRFileByName (pathname);

   RFileSystem_preDelete (rf);
   RFileSystem_observeExists (rf);
   result = DeleteFileA (pathname);

   if (result) RFileSystem_postDelete (rf);
   RFile_release (rf);

   return result;
}
```

Figure 18. Naccio/Win32 platform interface wrapper for DeleteFileA.

4.4 Expressiveness

The platform interface defines a set of resource operations by providing an operational specification for a system in terms of those resource operations. Altering the platform interface allows new resource operations to be defined. Hence, the range and precision of policies that can be defined is no longer limited by a standard set of resource descriptions. We can define new resource operations that correspond to any manipulation visible to the platform interface. The level of the platform interface limits what manipulations are visible, and thus the scope of policies that can be defined.

If the platform interface is at the level of a system API, we can define resource operations that correspond to any manipulation done through API calls. In the case of Naccio/JavaVM, the platform interface is at the level of the Java API. This means we can define a resource operation corresponding to any routine in the Java API. Since all manipulations of files, the network, display, threads, and the system environment are done through calls to the Java API, this supports a large class of policies. Some resources, however, are not manipulated through Java API routines, and cannot be defined using a platform interface at this level. For example, memory use is not done using the Java API. Some memory use is visible through Java API constructor calls, but memory use resulting from allocating arrays and constructing objects without using Java API constructors is not visible through Java API calls. If we wish to support policies defined using a memory resource, a lower level platform interface is required. This could be done either using callbacks from a modified virtual machine or by inserting resource operation calls that represent memory use into the application.

The platform interface also places fewer constraints on what can be done around a constrained event. In both the Naccio/JavaVM and Naccio/Win32 platform interface languages, there are no restrictions on the code that may be used in a wrapper. This means the behavior of the program may be changed in radical ways at any execution point visible to the platform interface. For example, we could write a wrapper for the `java.net.Socket` constructors that opens a window that plays Tetris and requires the user to accumulate a certain number of points before a socket is opened. More practical policies that take advantage of the extensibility of the platform interface might log all network transmissions to a secure audit file or make all windows created by an untrusted program appear with a red title bar.

Chapter 5
Compiling Policies

All policies that can be defined using the Naccio definition mechanisms can be enforced on executions. Policy enforcement mechanisms are divided into two phases – policy compilation prepares what is needed to enforce a policy on any program, and program transformation prepares a modified version of a target program that is constrained by a policy. This chapter discusses the policy compilation phase.

The policy compiler takes a policy description consisting of resource descriptions, a platform interface, and a resource use policy, and produces a policy description file that compactly specifies what transformations are needed to enforce the policy, as well as supplementary files used in those transformations. Those supplementary files include implementations of the resource operations that perform the checking specified by the policy. For platform interfaces at the level of a system API, they also include a modified system library that calls the relevant resource operations as directed by the platform interface.

Policy compilation is divided into three steps:

1. Processing the resource use policy to weave checking code into an intermediate representation of the resource operations (described in Section 5.1),
2. Reading the platform interface and analyzing it in conjunction with the resource operations (described in Section 5.2), and
3. Generating output files from the intermediate representations. For platform interface at the level of a system library, the output files comprise a policy-enforcing library that can be used in place of the standard system library to enforce a safety policy on an execution. The policy-enforcing library consists of implementations of the resources that incorporate checking code defined by the policy (described in Section 5.3), and a wrapped version of the standard library that calls routines that correspond to the abstract resource operations (described in Section 5.4).

Section 5.5 discusses some opportunities for optimizations involving both the resource implementations and library wrappers. The final output of the policy compiler is a policy description file that encodes the transformations needed to enforce the policy on a particular program (described in Section 5.6).

5.1 Processing the Resource Use Policy

The first step in compiling a policy is to parse the resource descriptions and resource use policy and produce an intermediate language representation of the checking code. This step is independent of the target platform and platform interface level. Hence, it can be reused by all Naccio implementations.

For the prototype implementations, the intermediate language is an abstract syntax tree similar to the Java programming language. This makes parsing the resource descriptions and resource use policy straightforward, and makes it easy to generate Java implementations from the intermediate representation. The disadvantage of using such a high level intermediate representation is that it may be harder to do certain optimizations at this level. For an industrial implementation, it may be better to use a lower-level intermediate representation or run an optimizer on the generated code.

Once the resource descriptions and resource use policy have been parsed, each safety property is instantiated with the constant arguments given in the resource use policy. These values are bound in the code by textually replacing instances of the parameter in the code with the actual parameter. If the same safety property is instantiated more than once in the policy with different arguments, multiple copies of the property will exist for each with different values bound to the parameters. Once the properties have been instantiated, the checking code associated with each safety property and required state block is integrated into the appropriate resource operations. State block helpers are merged into the resource class as methods. A copy of the checking code is inserted into the code body of each resource operation or group listed in the check clause. The code must be located in the body according to its type: all `precode` blocks in state blocks must be executed before any other checking code; all check clauses in permissions must be before safety property check classes since the allowance must overrides a violation by calling `allow` before the violation is reached; and all `postcode` blocks in state blocks must be executed after all checking code. To support this, the policy compiler maintains four separate code blocks for each resource operations corresponding to code from `precode` blocks, code from permission check clauses, code from safety property check clauses, and code from `postcode` blocks. Once all the safety properties have been processed, the code from each of these blocks is merged into a single block. Checking code preserves information about the property it came from. This information is used in the code generation phase so that violation messages can be produced that include information about the property that produced a violation.

Next, a relaxation algorithm is use to determine which resource operations, helpers and groups do meaningful work. Since a policy may require generic state blocks, but not use all state maintained by the block in checking, it is possible that some resource operations do not need to be implemented. This analysis is also a useful test that the policy means what the policy author intends as Naccio provides information on what resource operations are implemented. For example, if a policy is designed to restrict access to files but the policy compiler reports that it does not need to implement `RFileSystem.openRead`, the policy author should suspect something is wrong with the policy definition.

The policy compiler determines which resource operations are unnecessary using a specialization of standard compiler optimization for dead code elimination [Aho86, p. 595]. Because the definition of useful code in a safety policy is narrow, we can eliminate more code then could be eliminated by a generic compiler. A resource member does meaningful work if any of the following are true:

1. It could issue a violation. This is assumed to be the case if its body calls the `violation` command or calls a helper method that could issue a violation. A more involved analysis could attempt to determine if the violation could ever in fact be issued by analyzing the code logic more deeply. Resource operations and constructors that only call the `allow`

command do not need to be implemented, since this is only meaningful if a violation could be issued. Resource helpers that call the `allow` command need to be implemented if they are called by a resource operation that could issue a violation.

2. It sets the value of some resource state that is meaningful. State is meaningful if its value is used in a meaningful resource member.
3. It is contained in the group list of a meaningful resource group.

The relaxation works by first assuming all resource state and members are meaningless, and iterating the definition of meaningful work through each resource member. The iteration continues until no new meaningful resource members are marked. It is guaranteed to terminate since each iteration either marks no new resource members as meaningful and leads to termination or marks a previously meaningless resource member as meaningful. The number of resource members is an upper bound on the number of iterations. In practice, only a few iterations are needed for most policies.

5.2 Processing the Platform Interface

The platform interface is defined using a platform-specific variant of the platform interface specification language. Hence, each Naccio implementation must provide a platform-specific parser that converts the platform interface to an intermediate representation. The platform interface intermediate representation is similar to that used for resource implementations. This allows much of the analysis code to be reused. Each platform interface wrapper is associated with some concrete system event and contains wrapper code for that event. For platform interfaces at the level of a system API, each wrapper is associated with a call to a system API routine.

A wrapper is considered to be a *normal form* wrapper if it always invokes the original wrapped operation exactly once and all wrapper code is limited to calling resource operations, setting wrapper state, doing side-effect free computation that is guaranteed to terminate, and calling helper functions and Naccio library routines that satisfy these properties. It is likely that there are many unnecessary platform interface wrappers, since the platform interface is written to support a large class of policies.

As with resource operations, the policy compiler uses a specialization of the standard compiler optimization for dead-code elimination to eliminate unnecessary normal form wrappers. A normal form wrapper is necessary if it either:

1. Calls a meaningful resource operation (as was determined by processing the resource use policy), or
2. Sets some meaningful wrapper state. Wrapper state is meaningful if it is read in a necessary wrapper.

Which wrappers are necessary is determined by a relaxation analysis similar to that used to determine which resource members are meaningful. Within a necessary wrapper, calls to resource operations that are not meaningful are removed.

In addition to what can be determined by the analysis, the policy compiler uses `requiredif` clauses to eliminate wrappers that could not otherwise be determined to be unnecessary. The policy compiler trusts the `requiredif` clause, and will eliminate a wrapper that has a `requiredif` clause if none of the resource operations listed do meaningful work. This also allows wrappers that are not expressed in normal form to be eliminated. Naccio cannot eliminate wrappers that are not normal form wrappers since determining that they use no meaningful resource operation and set no meaningful state is not sufficient. Non-normal form wrappers can also change the return value, call routines that alter the behavior of the program, or prevent the original routine call from occurring. Hence, wrappers that are not normal form are never eliminated except when permitted by an explicit requiredif clause.

One example is the checked stream classes used in the Java API platform interface for java.net.Socket. The wrapper for getOutputStream creates a new NCheckedNetworkOutputStream object that extends the result from the original method and overrides the write methods to perform checking code before calling the superclass method. All this work is unnecessary unless the RNetwork.preSend or RNetwork.postSend operation does meaningful work. Because the wrapper has a requiredif clause that indicates this, Naccio/JavaVM can eliminate the wrapper and the helper class if the RNetwork.preSend and RNetwork.postSend operations are not meaningful.

5.3 Generating Resource Implementations

The intermediate representations of the processed resource operations need to be converted to implementations that perform the actual checking. A resource implementation must be produced for every resource that contains a meaningful resource member. The code produced depends on the target platform, but some platform-independent transformations can be done on the intermediate representation first.

The violation and allow commands in safety policy bodies are replaced with calls to Naccio library routines. The library routines take extra arguments giving the names of the policy and property that issued the violation and information on where it is defined. For certain policies, the violation and allow library routines also need an extra argument that encodes the violation status code. This is necessary if the policy uses weaken to combine permissions and negative properties since violation codes are used by allow command to override future violation commands. If violation codes are necessary, a parameter of type ViolationCode is added to all resource members that call the violation or allow command. The Naccio library defines the ViolationCode type. It encodes whether an allow command was issued that should suppress violations detected in this resource operation. A ViolationCode object is created in the wrapper routing and passed to resource operations. The policy compiler adds parameters to declarations and inserts them at call sites as necessary. The ViolationCode object is passed to the allow and violation library methods. The allow method sets it to record a permission, and the violation method uses it to suppress violations that have been overridden by permissions.

The other preparation step is to handle resource groups. For each resource group, there are two implementation options: we can implement it as a method helper and add calls from the group members, or we can inline the checking code directly into group members. We must pay attention, however, to the appropriate ordering of checking code. In the worst case, this means a resource group implementation is divided into four separate helper routines corresponding to the precode actions, the permission (allow) bodies, the negative check bodies, and the postcode actions. Group members must call each of these at the corresponding point in their own check body. Fortunately, for most resource groups only one or two of the routines are necessary. Implementing resource groups as methods saves code duplication, but involves the overhead of up to four additional method calls for each group member. The group member list gives the arguments necessary to call the resource group. This is converted into the intermediate representation of the equivalent method call. An alternative is to inline the group code directly into the member bodies. For simplicity, this is only done for resource members that match the group parameters exactly. It could be done for other members, but this would require binding the group parameters to new local variables.

A further improvement is possible if the individual group member has no checking code other than that given by the resource group. For group members that have no checking code other than that done by the resource group, we can directly replace the member with the group and avoid the overhead of either extra implementation or method calls. We simply implement the resource group as if it were an operation, and replace calls to the resource operation in the platform interface with calls to the group.

Finally, a platform-specific implementation of the resources is generated. The actual implementation depends on the particular target platform, but generating resource implementations from the intermediate representation should be relatively straightforward for most platforms. The next two subsections discuss how each prototype implementation generates resource implementations.

5.3.1 Naccio/JavaVM

Naccio/JavaVM generates a Java class corresponding to each resource. Java source code is produced and compiled using a standard Java compiler. Since the intermediate representation is similar to Java source code, producing source code for the corresponding Java class is straightforward.

Figure 19 shows the resource class for the RFileSystem resource description from Figure 4 that was generated by Naccio/JavaVM to enforce the LimitWrite policy introduced in Figure 7. This file is placed in a newly created output directory corresponding to a new package holding all the resource implementations for this policy. Because RFileSystem is declared as a global resource, all class variables and routines are static. The bytes_written field introduced by TrackBytesWritten is implemented by adding a class variable to RFileSystem.[11]

The modifyExistingFile method corresponds to the group with the same name and contains code from the NoBashingFiles property. The violation command has been converted to a call to the NCheck.policyViolation library method, and additional arguments are passed so a helpful error message can be produced. Since none of the members of the modifyExistingFile group have their own checking code, modifyExistingFile can be implemented as a method if calls to group members in the platform interface are replaced in the generated platform interface wrappers with calls to modifyExistingFile. The preWrite and postWrite methods contain code from the LimitBytesWritten safety property. The limit parameter of LimitBytesWritten has been bound to the value of 1000000 passed in by the LimitWrite property.

The implementation shown does not pass violation codes since the policy did not use permissions. If violation codes were necessary, each resource routine would have an additional parameter of type naccio.library.ViolationCode and would pass this parameter on to the policyViolation library method and a similar method corresponding to allow.

---

[11] The type of bytes_written is long. Strictly, it should be a Naccio library type with the semantics for the int type defined for use in safety properties. For simplicity, the Naccio/JavaVM implementation ignores issues of precise number semantics (such as integer overflow), and assumes using a long to represent unbounded integers is sufficient.

```
package lw; // Note: actually a longer, unique package name is used.  For readability we shorten it here.
import naccio.library.*;

public class RFileSystem {
   static long bytes_written = 0; // from TrackBytesWritten
   final public static void modifyExistingFile (lw.RFile file) {
      naccio.library.NCheck.policyViolation (“LimitWrite”, "NoBashingFiles",
                                                  "Destructive manipulation of file: “ + file.getName ());
   }

   final public static void preWrite (lw.RFile file, long n) {
      if (bytes_written + n > 1000000)
         naccio.library.NCheck.policyViolation (“LimitWrite”, "LimitBytesWritten",
            "Attempt to write more than " + 1000000 + " bytes. Already written " + bytes_written +
            " bytes, writing " + n + " more to " + file.getName () + ".");
   }

   final public static void postWrite (lw.RFile file, long n) {
      bytes_written += n;
   }
}
```

Figure 19. Resource class generated by Naccio/JavaVM.

The generated Java source files are compiled by running a standard Java compiler. The resulting class files are then transformed to replace calls to wrapped API routines with calls to the corresponding unwrapped API routines. This is done using the same transformation engine and similar transformations as is used to produce the platform interface (see Section 5.4). The calls to wrapped API routines are rewritten so that checking is not done for API calls made in the resource implementations.

5.3.2 Naccio/Win32[12]

Naccio/Win32 generates resource operations as ANSI C source code that is compiled into a DLL. ANSI C is chosen as the implementation language instead of C++ because of portability issues and simplicity, and over other languages because of efficiency and the ease with which a DLL can be produced from C source code. Since C is not object-oriented, a naming convention is used to group routines associated with a particular resource and the associated resource object is passed explicitly. Macros are used to hide these implementation details from the platform interface.

Naccio/Win32 produces both a header file and source file containing all the resource implementations. The header file contains type definitions, variable and function declarations, and macro definitions that are used in the platform interface implementation. Both the resource source file and the platform interface implementation source file include this header file.

The resource header generated by Naccio/Win32 for the LimitWrite policy is shown in Figure 20. The types for RFileSystem and RFile are defined as pointers to structures containing fields that correspond to the resource state. Since RFileSystem is a global resource, the resource header file also declares the variable RFileSystem_state of type RFileSystem to represent the global RFileSystem object. This simplifies the implementation of resource operations, since it allows

---

[12] This section is based on [Twyman99], which contains additional information on how resource implementations are generated by Naccio/Win32.

global and non-global resource operations to be implemented identically except the global state object is passed instead of the this object. For an industrial implementation, it would make more sense to put the state associated with global resources in stand-alone variables instead of structure types would save the overhead of passing an extra pointer and performing an extra indirection.

The header file defines empty macros for the resource operations that do no useful work. Since macros are expanded at compilation time, this means the resource calls can be left in the platform interface with no run-time overhead. The header file also defines macros for the modifyExistingFile group member operations that call RFileSystem_modifyExistingFile with the appropriate argument. For resource operations that do useful work, the resource header file includes macro definitions that automatically pass the global state to the actual resource operation implementation.

```
#ifndef _INSIDE_RESOURCE_DLL_
#define NACCIO_RESOURCE DLLIMPORT
#else
#define NACCIO_RESOURCE DLLEXPORT
#endif

typedef struct _RFileSystem { long bytes_written;} *RFileSystem;
typedef struct _RFile { String name; } *RFile;

NACCIO_RESOURCE extern RFileSystem RFileSystem_state;
NACCIO_RESOURCE RFileSystem RFileSystem_new();

#define RFileSystem_initialize()            /* empty macro body */
#define RFileSystem_terminate()             /* empty macro body */
#define RFileSystem_openRead(p_a0)  /* empty macro body */
… /* Similar no-op's for other resource operations elided */

#define RFileSystem_openOverwrite(p_a0)    RFileSystem_modifyExistingFile (p_a0)
#define RFileSystem_openAppend(p_a0)       RFileSystem_modifyExistingFile (p_a0)
#define RFileSystem_preDelete(p_a0)        RFileSystem_modifyExistingFile (p_a0)
#define RFileSystem_rename(p_a0,p_a1)      RFileSystem_modifyExistingFile (p_a0)

NACCIO_RESOURCE void RFileSystem_op_preWrite(RFileSystem p_this, RFile p_file, long p_n);
#define RFileSystem_preWrite(p_a0,p_a1) \
            RFileSystem_op_preWrite (RFileSystem_state, p_a0, p_a1)

NACCIO_RESOURCE void RFileSystem_op_postWrite(RFileSystem p_this, RFile p_file, long p_n);
#define RFileSystem_postWrite(p_a0,p_a1) \
            RFileSystem_op_postWrite (RFileSystem_state, p_a0, p_a1)

NACCIO_RESOURCE void RFileSystem_op_modifyExistingFile(RFileSystem p_this, RFile p_file);
#define RFileSystem_modifyExistingFile(p_a0) \
            RFileSystem_op_modifyExistingFile (RFileSystem_state, p_a0)

NACCIO_RESOURCE RFile RFile_new(String p_pathname);
NACCIO_RESOURCE void RFile_delete(RFile p_this);
NACCIO_RESOURCE String RFile_getName(RFile p_this);
```

Figure 20. Resource headers file generated by Naccio/Win32.

```
#include <naccplat.h>
#define _INSIDE_RESOURCE_DLL_
#include "resource.h"

NACCIO_RESOURCE  RFileSystem RFileSystem_state;

NACCIO_RESOURCE RFileSystem RFileSystem_new () {
   RFileSystem p_this = nAlloc (sizeof (struct _RFileSystem));
   p_this->bytes_written = 0;
   return (p_this);
}

NACCIO_RESOURCE void RFileSystem_op_modifyExistingFile (RFileSystem p_this, RFile file) {
  String tempstr_0 = NULL, tempstr_1 = NULL, tempstr_2 = NULL, tempstr_3 = NULL;
  Check_policyViolation (String_fromlit (&tempstr_0, "LimitWrite"),
      String_fromlit (&tempstr_1, "NoBashingFiles"),
      String_concat (String_concat (String_empty (&tempstr_2),
                                         String_fromlit (&tempstr_3, "Destructive manipulation of file:")),
                       RFile_getName(file)));
   String_delete(tempstr_0); String_delete(tempstr_1); String_delete(tempstr_2); String_delete(tempstr_3);
}

NACCIO_RESOURCE void RFileSystem_op_preWrite (RFileSystem p_this, RFile file,  long n) {
   if (RFileSystem_state->bytes_written + n > 1000000) {
      Check_policyViolation (…); // Lots of ugly string manipulation code elided
   }
}

NACCIO_RESOURCE void RFileSystem_op_postWrite (RFileSystem p_this, RFile file, long n) {
   RFileSystem_state->bytes_written += n;
}

… // Construction and destruction functions for RFile elided.

BOOL APIENTRY DllMain (HANDLE hMod, DWORD ul_reason_for_call, LPVOID lpRes) {
   switch (ul_reason_for_call) {
      case DLL_PROCESS_ATTACH:   RFileSystem_state = RFileSystem_new ();
                                 RFileSystem_initialize (); break;
      case DLL_PROCESS_DETACH:   RFileSystem_terminate (); break;
   }
   return (TRUE);
}
```

Figure 21. Implementation resource.c generated by Naccio/Win32 for LimitWrite.

The policy compiler produces implementations in a source file, resource.c, shown in Figure 21. This file includes the resource.h header file. Implementations of resource operations are generated from the intermediate representations. The main complication is dealing with the library String type, since C does not provide a useful string datatype. The generated code declares temporary string variables for use in concatenating strings. The strings must be passed to String_delete before the function returns to reclaim memory used by the string.

This file also defines the DllMain function, which is called when the DLL is attached or detached from a process. Since the resource DLL is implicitly linked by the API wrapper DLL, this function will be called at the beginning of execution. When the DLL is attached, it initializes RFileSystem_state to a new RFileSystem object and calls the RFileSystem_initialize operation. When the program shuts down, it calls detach for each implicitly linked DLL. This will call the terminators in the DLL_PROCESS_DETACH case of DllMain.

5.4 Generating Platform Interface Wrappers

In addition to the resource implementations, the policy compiler must produce an implementation of the platform interface. What is involved depends on the level of the platform interface. For a hardware-level platform interface, it would involve building traps into the hardware device's system software and writing support code necessary to obtain enough information to call the appropriate resource operation. For a platform-interface at the level of machine instructions it involves performing low-level transformations on the object files to introduce platform interface wrapper code where appropriate. A different approach would be to write an interpreter that executes the program and runs the relevant wrapper before interpreting a wrapped instruction. Our focus is on platform interfaces at the level of a system API, and for the remainder of this section we assume the platform interface is at that level.

Most of the platform interface generation is platform specific, but some work is done processing the intermediate representation first. If the policy needs violation codes, the intermediate representations are modified to introduce them. A single `ViolationCode` object should be maintained throughout the wrapper body and passed to each resource operation. A local variable declaration is inserted at the beginning of the wrapper to store this object, and it is assigned to the result of a library creation routine. This object is inserted at the beginning of the parameter list for calls to resource operations. Between resource operations, its value is reset since allowances do not carry over resource operations. At the end of the wrapper routine, the library `ViolationCode.release` routine is called. This supports the possibility for handling `ViolationCode` objects on a platform that does not support garbage collection, such as Naccio/Win32. It is also useful since in conjunction with the creation routines it allows `ViolationCode` objects to be reused and avoids the costs associated with creating many objects with short lifetimes.

Once the intermediate representations of the wrappers have been produced the next step is to convert them to a form that can be easily integrated into a program to enforce the policy. There are two approaches: modifying the system API itself or interposing wrapper code between the program and the system API. The first approach offers more flexibility in controlling the interactions between wrapper and system code but requires lower-level manipulations of object files. Naccio/JavaVM modifies system API code while Naccio/Win32 uses an interposition layer that performs the necessary checking.

5.4.1 Naccio/JavaVM

Platform interface wrappers for Naccio/JavaVM are implemented by rewriting Java API class files. For each policy, Naccio/JavaVM creates a (possibly partial) copy of the Java API classes that are altered to implement the platform interface wrappers. The transformation engine is based on JOIE, a toolkit for manipulating Java class files [Cohen98]. Information on JOIE and other transformation engines is found in Section 7.4.

Java binary compatibility

Rewriting classes depends on being able to run the original program with modified library classes without recompiling. The Java Language Specification [Gosling96, Chapter 13] describes changes that can be made to class files without breaking link compatibility in conforming Java virtual machines. Compatible changes include adding new fields, methods or constructors to an existing class or interface and changing the implementation of existing methods, constructors and initializers. All the class modifications done by Naccio/JavaVM are designed to preserve binary compatibility.

Java binary compatibility is not guaranteed in the presence of native methods and Java implementations are expected to describe binary compatibility of native methods.[13] This poses a problem for Naccio/JavaVM, since it may need to modify classes used by native methods and does not necessarily have access to the source code for the native method. As a result, supporting binary compatibility across native methods depends on a particular JavaVM implementation. The prototype Naccio/JavaVM implementation assumes that binary compatibility holds for inserting fields, methods and constructors and replacing routine implementations even in the presence of native methods. This is in fact not the case for Sun's JDK 1.1 Java implementations, since adding new fields can interfere with field referencing.[14] It is believed that JDK 1.2 and future implementations will not have this problem, although no formal claims about binary compatibility across native methods are made by the JDK 1.2 documentation [Kramer99].

Wrapping classes

To produce a wrapped version of an API class, the policy compiler alters the class byte codes to reflect the state and wrappers defined by the platform interface. State defined in the platform interface wrapper is implemented by adding fields to the class. These fields are declared private, since they may only be used in the platform interface wrappers.

To wrap a method, the wrapper code from the platform interface is translated from the intermediate representation into Java byte codes and inserted into the class file in place of the original method. The original method is renamed by adding a prefix (`o_`) to the method name. Since no methods in the Java API start with `o_`, this always produces a unique name. Renaming the original method implementation allows the wrapped version of the method and other routines in the class library to call the original method.

The hash marker in the platform interface wrapper is replaced with a call to the original method. If it has a return value, the result is stored in a new frame location that corresponds to the `result` local variable in the platform interface wrapper. At exit points of the wrapper, this result is returned. Note that exceptions produced in the original method call will propagate directly through the wrapper code. This means the checking code after the hash marker will not execute if the original method call throws an exception. For most of the Java API, this is probably the correct semantics since the resource manipulation does not occur if the API method throws an exception. In some cases, some API methods will do partial resource manipulations before throwing an exception. Platform interface authors can use a `catch` statement around the hash mark to implement appropriate resource calls after the exception, and then re-throw the exception.

Constructors and native methods introduce a few complications. Since the class determines the names of constructors, we cannot rename them. Instead, we add an extra argument to distinguish the original constructor from any other constructors. This means when Naccio transforms wrapped routines to call the unwrapped version of a wrapped constructor, it must push an extra

---

[13] According to the Java language specification, "The impact of changes to Java types on preexisting native methods that are not recompiled is beyond the scope of this specification and should be provided with the description of an implementation of Java. Implementations are encouraged, but not required, to implement native methods in a way that limits such impact."

[14] This was discovered through experimentation and code analysis. There is in fact no documentation that describes the binary compatibility rules for Sun's JDK implementations.

argument on the stack and change the type descriptor of the constructor it calls. Adding the extra argument to distinguish the original constructor simplifies the work that will be needed to transform an application to call the wrapped constructors. The type of the extra argument is chosen so that the new constructor does not conflict with any existing constructor. Since application code always calls the wrapped constructor, there is no need for the program transformer to alter constructor calls in application classes.

For native methods, we cannot change the method name since the JavaVM will not be able to find the corresponding native method implementation. Instead, we introduce a new method (named w_*method*) that implements the wrapper and calls the original native method. This means the program transformer will need to replace calls to wrapped native methods in the application with calls to the corresponding w_*method*. An alternative would be to rename the native method and modify the VM so that it can still map the new name to the correct native method. This would eliminate the need to change wrapped native method names in application classes, but would not be portable across different VM implementations. Using a new name for wrappers for native methods means we need to replace calls to the native method in application and unwrapped library code with calls to the new wrapped method instead. We could handle all methods this way and rename non-native methods also. This would make the policy compilation and transformation process simpler and more consistent. This is not done, however, since it would involve extra work at transformation time since applications must be modified to call the w_*method*s instead of the unwrapped methods. Since program transformation is done much more frequently than policy compilation, we prefer to add a little complexity to the policy compiler to reduce the time required to transform an application.

Pass-through checking

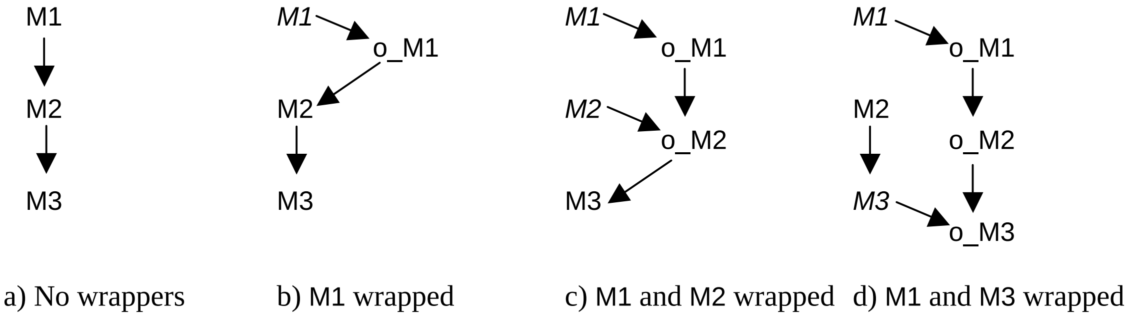
The tricky part of rewriting the library classes is supporting the pass-through semantics correctly. The semantics required by the Naccio/JavaVM platform interface are:

- Calls to Java API routines in the bodies of pass-through routines should call the wrapped versions of those routines.
- Calls to Java API routines in the bodies of regular wrapped routines should call the unwrapped versions of those routines.

Wrappers must pass through recursively – if a wrapped routine calls an API routine that has no explicit platform interface wrapper, but that calls a wrapped API routine we must ensure that it calls the unwrapped version.

Consider the simple dependency graph shown in Figure 22a. The body of method M1 calls method M2 and the body of M2 calls M3. Figure 22b depicts the situation where M1 has a regular wrapper. Naccio/JavaVM produces a copy of M1 named o_M1 that is the original implementation of M1 and replaces M1 with a new M1 method that implements the wrapper code and calls o_M1.

Figure 22c shows the scenario where M2 has a wrapper also. As before, Naccio/JavaVM produces o_M2 as the unwrapped version of M2. Since the regular wrapper for M1 is intended to account for all meaningful resource manipulations done by M1, it should call the unwrapped version of M2. After wrapper generation, the transformation engine replaces the call to M2 in the body of o_M1 with a call to o_M2.



a) No wrappers b) M1 wrapped c) M1 and M2 wrapped d) M1 and M3 wrapped

Figure 22. Pass-through semantics.

Wrapped methods are shown in italics, original methods are renamed o_M.

In Figure 22d, M1 and M3 have wrappers but M2 does not. We need to ensure that the indirect call to M3 from the wrapped M1 calls the unwrapped o_M3 instead of the wrapped M3. Otherwise, the wrapper checking code associated with M3 would be executed when o_M1 calls M2, which then calls M3. To provide the necessary semantics, an internal version of M2, o_M2, is introduced for M2. It contains a copy of the M2 code, but calls to M3 are replaced with calls to o_M3. This allows implementations of wrapped routines to call the unwrapped versions of nested routines. An internal version of a routine is necessary if it calls a wrapped routine and it is called by a wrapped routine. Direct calls to M2 pass through checking normally, since it calls the wrapped M3 method normally.

Superclass methods

Another situation Naccio/JavaVM must deal with is where a platform interface wrapper is provided for a subclass method that overrides a (possibly abstract) method in a superclass. For example, consider the situation if there is a wrapper for java.io.FileOutputStream.write. The class FileOutputStream is a subclass of OutputStream, and OutputStream declares write as an abstract method. We must ensure that application calls to OutputStream.write on objects that are FileOutputStream types call the wrapped version of write, but calls on objects that are not of type FileOutputStream call the appropriate unwrapped write method. Since FileOutputStream.write is a native method, the wrapper method w_write is added to FileOutputStream. We need to provide a w_write method for OutputStream also, so application classes can be rewritten to call w_write on OutputStream objects that are not necessarily FileOutputStream objects. The policy compiler inserts the w_write method into OutputStream. Its body simply calls write with the same arguments. If the w_write is called on a FileOutputStream object, it will dispatch to the subclass method that is the wrapped version of write. If it is called on an OutputStream object of a type that does not wrap write, the w_write method added to OutputStream will call the regular write method.

Similarly, if a non-native subclass method overrides an unwrapped method and renames the original method o_*method*, Naccio/JavaVM adds an o_*method* to the superclass. Its implementation calls *method* with the original arguments. This allows calls in wrapped API methods that should call the unwrapped version of the method, to call o_*method* regardless of the subtype.

Renaming classes

There are two different ways to generate the policy-enforcing wrapper classes. The simplest way is to write the modified class files in a new directory and use the Java CLASSPATH to select the policy-enforcing library when an application constrained by the policy runs. If we wish to support multiple policies in the same Java VM, we need a way to identify the API classes of each policy-enforcing library at run-time. This is done by globally renaming all classes in the policy-enforcing version of the API to include a unique package name so that they can be identified (e.g., java.io.File becomes policy_lw.java.io.File). To rename classes consistently, all classes in the API must be renamed. All references to the API classes are replaced with the policy-enforcing class names.

Stripping SecurityManager calls

In addition to inserting the Naccio checking calls, the policy compiler can be used to remove calls to the JDK SecurityManager. Note that this is done only because the JDK security mechanisms are built into the Java API. To remove the run-time overhead associated with them to enable the performance analysis in Section 8.4 the policy compiler can be directed to strip these calls from the API classes. In the Sun JDK 1.1 implementation, all security manager calls involve either calling System.getSecurityManager method to obtain the security manager or using the private security instance variable in the java.lang.System class. This is followed by a comparison to null with a branch that calls a security manager check method. The policy compiler can recognize the sequence and remove the code associated with obtaining a security manager, testing if it is null, and calling a check method.

Example

Figure 23 shows the policy-enforcing library class generated for java.io.FileOutputStream by Naccio/JavaVM to enforce the LimitWrite policy. The actual contents of the class files are simplified for readability, but are essentially what is shown here.

The top of each class file shows the visible declarations in the class. The policy-enforcing version contains an extra field declaration corresponding to the rfile state defined in the platform interface for java.io.FileOutputStream (see Figure 11). The rewritten class defines several methods and constructors not defined in the original class. Because the LimitWrite policy attaches checking code to the resource operation associated with overwriting an existing file, and every constructor in FileOutputStream may open a file for overwriting, every constructor needs a wrapper. Hence, for every constructor there is a new constructor declaration with a dummy argument added to distinguish it from the original. This argument can be any type that does not lead to a conflict with an existing constructor. In this case, none of the constructors have arguments of type short and Naccio/JavaVM uses short for the type of the dummy argument.

The code for the constructor taking a java.io.File object is shown. In the original class, it calls the constructor taking a String, passing in the absolute path of the file object. In the rewritten class, there are two versions of the constructor. The unwrapped version takes an extra parameter of type short to distinguish it from the wrapper version. Its body is copied from the body of the original constructor, except that the call to the String constructor is replaced with a call to its unwrapped version by adding a dummy argument. Otherwise, the wrapper would call the wrapped version of the constructor and execute inappropriate checking code. The wrapped version of the constructor incorporates the code from the platform interface. It calls the doOpen helper method, then calls the unwrapped version of the constructor, and finally stores the RFile object in the rfile instance variable. The implementation of doOpen is based on its code in the platform interface. The call to RFileSystem.openOverwrite has been replaced with a call to the resource group method RFileSystem.modifyExistingFile. This can be done since openOverwrite has no checking associated with it except for what is done by the modifyExistingFile group. The

public class FileOutputStream	public class FileOutputStream
extends OutputStream {	extends OutputStream {
	private lw.RFile rfile;
public FileOutputStream(String);	**public FileOutputStream(String);**
	public FileOutputStream(String,short);
public FileOutputStream(String,boolean);	**public FileOutputStream(String,boolean);**
	public FileOutputStream(String,boolean,short);
public FileOutputStream(File);	**public FileOutputStream(File);**
	public FileOutputStream(File,short);
public FileOutputStream(FileDescriptor);	**public FileOutputStream(FileDescriptor);**
	public FileOutputStream(FileDescriptor,short);
public native void write(int);	public native void write(int);
	public void w_write(int);
public void write(byte[]);	**public void write(byte[]);**
	public void o_write(byte[]);
public void write(byte[], int, int);	**public void write(byte[], int, int);**
	public void o_write(byte[], int, int);
public native void close();	public native void close();
public final FileDescriptor getFD();	**public FileDescriptor getFD();**
	public final FileDescriptor o_getFD();
protected void finalize();	protected void finalize();
	public static lw.RFile doOpen(File);
}	}
Method FileOutputStream(File)	*Method FileOutputStream(File,short)*
FileOutputStream(getPath <arg 1>)	FileOutputStream (getPath <arg 1>, 0)
	static Method lw.RFile doOpen(File)
	<local 1> := lw.RFileMap.lookupAdd (<arg 1>)
	if o_exists(<arg 1>)
	lw.RFileSystem.modifyExistingFile (<local 1>)
	areturn <local 1>
	Method FileOutputStream(File)
	<local 1> := doOpen(<arg 1>)
	FileOutputStream(<arg 1>, 0)
	rfile := <local 1>
Method void write(byte[])	*Method void o_write(byte[])*
writeBytes (<arg 1>, 0, <arg1>.length)	writeBytes (<arg 1>, 0, <arg1>.length)
	Method void write(byte[])
	if rfile != null
	lw.RFileSystem.preWrite (rfile, <arg 1>.length)
	o_write (<arg 1>)
	if rfile != null
	lw.RFileSystem.postWrite (rfile, <arg 1>.length)
// no code for native void write (int);	**Method void w_write(int)**
	if rfile != null
	lw.RFileSystem.preWrite (rfile, 1)
	write(<arg 1>)
	if rfile != null
	lw.RFileSystem.postWrite (rfile, 1)

Figure 23. Generated policy-enforcing library class for java.io.FileOutputStream.

Left side shows the original API class. Right side shows the rewritten class file using the LimitWrite safety policy. Classes are simplified and excerpted for clarity. Renamed original routine declarations are shown in *italics*, wrappers are shown in **bold**.

call to RFileSystem.openCreate in the branch for creating a new file has been removed since there is no checking associated with the openCreate operation for the LimitWrite policy.

The figure also shows two write methods, one that writes an array of bytes and one that writes a single byte. The array of bytes method illustrates what is done for a normal wrapper. The rewritten class contains a method o_write(byte[]) that contains the original method body. The wrapped version uses the unmodified name and contains a body compiled from the platform interface. Its body calls o_write where the original method call marker was. Since the write(int) method is a native method, the original class contains no implementation for it. Renaming native methods is not possible, so the wrapper for write is named w_write. It body calls the unwrapped native write method. Application classes will be modified to call w_write(int) instead of write(int).

5.4.2 Naccio/Win32

Generating the platform interface wrappers for Naccio/Win32 is simpler than for Naccio/JavaVM since pass-through semantics are not supported. Further, the Win32 platform interface is written as a stylized C file that can be compiled directly. Once the resource header file has been generated, all that is necessary it to compile the platform interface file in a directory containing this header file. The compiler is run with the appropriate linker directives to forward references to null wrappers to the system DLL. The resulting DLL is renamed with a different extension so it can be distinguished from the system DLL by the loader.

In addition to compiling the platform interface file, Naccio/Win32 must generate a definition (.DEF) file that lists every function exported by the wrapper DLL. For a wrapped function, the export table contains an entry that maps the original function name to the name of the wrapper. For the wrapper for DeleteFileA shown in Figure 18, the corresponding export table entry is: DeleteFileA=wrapper__DeleteFileA. This makes calls to DeleteFileA in programs linked with the wrapper DLL call the wrapper__DeleteFileA function defined by the platform interface wrapper instead. Functions in the original DLL that are not wrapped can be listed as indirections in the import table. These will be replaced with calls to the original DLL at load time. Section 6.1.2 explains how the program transformer modifies an application to use the generated wrapper DLL.

5.5 Integrated Optimizations

All the optimizations discussed so far are done independently on either the resource implementations or platform interface wrappers. Information about which resource operations are meaningful is used to remove unnecessary resource operation calls from the platform interface, but otherwise all optimizations are done independently. Breaking the barrier between resources and platform interface wrappers offers the potential for additional optimizations. The prototype implementations do not perform any of the integrated optimizations discussed here, however they could be done by an industrial implementation that is concerned with the run-time performance of the transformed code.

Without integrated optimizations, users suffer the run-time overhead associated with policies being expressed at an abstract level. This includes the overhead associated with creating abstract resource objects and carrying out extra routine calls. The solution is to inline both routines and state. Inlining routines is a standard compiler optimization and can be done straightforwardly. Code from the resource operation can be moved into the wrapper. Since the resource code is usually small and most resource operations are only called a few times in the platform interface, inlining resource operations is almost always worth doing.

Inlining state is less traditional, since it depends on the limited semantics supported by the resource use policy. In certain situations that are quite common in typical platform interfaces, we can move resource state into the platform interface. This eliminates the need for resource objects and saves the overhead required to create, store and garbage collect resource objects as well as the overhead necessary to reference object fields. Inlining state can be done only if the identity of resource objects is irrelevant. If the resource objects are shared or compared as objects, inlining state would change the meaning of the platform interface. As it happens, most state fields in resource objects are immutable objects used only to store values. These fields can be safely inlined into the platform interface class that uses the resource object. Inlining resource objects would involve removing the resource objects and moving their instance variables directly into the associated application level object.

Further opportunities for integrated optimizations are possible if the application is analyzed also. Since this is likely to take a long time, it only makes sense for performance-critical applications that will be run frequently. Static analysis of the program text in conjunction with the safety policy analyses can be used to remove safety checking that is determined to never lead to a violation. For example, if the policy prohibits network connections except to hosts in a particular trusted domain, a static analysis could attempt to determine the remote address of all network connections opened by the program. If all addresses can be determined statically, and all are in the trusted domain, the checking code associated with opening network connections can be removed. After this is done, the relaxation analysis of the resource operations should be repeated since removing the checking code may have rendered more resource operations and state unnecessary.

The other optimization that can be done through static analysis of the program text is batching checking. For example, if there is checking code associated with the `RNetwork.preSend` operation and the program contains a loop that sends one byte at a time, then each send requires the overhead of calling a wrapper routine, calling the `preSend` operation, and executing its body code which does some checking and increments a state value. If the number of loop executions can be determined, the checking code can be moved out of the loop and `preSend` called once with a parameter that accounts for all network sends that will be done in the loop. If this call issues no violations, the entire loop can be executed without checking the send calls. This kind of optimization depends on knowing that calling `preSend (connection, n1)` and `preSend (connection, n2)` is the same as calling `preSend (connection, n1 + n2)`. This depends on the policy code associated with `preSend`. While it is unlikely that the policy compiler could determine this automatically, if Naccio were extended to support descriptive annotations a policy author could add annotations to document that this is the case and thereby enable the optimization.

It is expected that in most situations the run-time benefits of application-integrated policy optimization would not outweigh the substantial analysis time necessary to analyze an application and perform application-integrated policy optimizations. These optimizations are also complex and the potential for flaws in the analyses introduces new vulnerabilities. They may be more useful in conjunction with a proof-carrying code system (see Section 7.1) where the code distributor does the optimizations. The distributor would ship an optimized version of the program constrained by a published policy along with a condensed proof that the distributed program satisfies the policy, and the receiver would use a (hopefully) small and simple system to verify that the policy is satisfied. This only works if the receiver agrees to a standard policy used by the distributor, although the receiver could enforce additional properties on the code.

5.6 Policy Description File

The final output of the policy generator is the policy description file. This file contains transformation rules that compactly describe the changes the program transformer must perform. It contains a rule that identifies the location of the policy-enforcing library. Rules may also direct the program transformer to rename specific system calls (for example, Naccio/JavaVM must rename wrapped native methods), and to modify the application to call resource initializers before execution begins and to call terminators before execution terminates. The format of the policy description file is platform-independent, although its contents are likely to be highly dependent on the particular platform and Naccio implementation.

Chapter 6

Transforming Programs

The program transformer takes a policy description file and a program and produces a new program that behaves like the original program except it is guaranteed to satisfy the safety policy used to produce the policy description file. The level of the platform interface establishes the extent of the program that is handled by the program transformer. The policy compiler is responsible for parts of the program described by the platform interface; the program transformer handles everything that is not described by the platform interface. For this chapter, we assume the platform interface is at the level of a system API.

In the modified program, calls to system API functions are replaced by calls to the appropriate policy-enforcing wrappers. For platforms where the system API is linked dynamically, it is often possible to do this by making some simple changes to the program executable or by setting parameters to the execution environment. Section 6.1 describes how Naccio/JavaVM and Naccio/Win32 replace system calls with calls to policy-enforcing wrappers.

In addition, the program transformer must ensure the integrity of the checking code either by modifying the program or by verifying that the necessary properties are satisfied. What actually must be done depends on the execution platform and on how the platform interface wrappers and resource implementations are implemented. At a minimum, the program transformer must ensure that hostile programs cannot circumvent safety checking by manipulating resources without going through the appropriate platform interface wrapper or by modifying checking code or data. Section 6.2 discusses what is necessary to guarantee the integrity of the checking for the JavaVM and Win32 platforms.

6.1 Replacing System Calls

Replacing system calls involves determining what code is part of the application program and altering the system calls it makes so the policy-enforcing wrappers are called instead. This can be done by renaming libraries, classes or routines, or by changing the execution environment. Since the system API is accessed differently depending on the execution platform, the solutions for Naccio/JavaVM and Naccio/Win32 are different. Both involve switching which API implementations are linked with the program.

6.1.1 Naccio/JavaVM

Naccio/JavaVM provides two different alternatives for replacing Java API classes. One option is to leave the application unchanged and set the `CLASSPATH` so that the modified classes are found before the standard Java API. An application request for an API class will transparently load the policy-enforcing version of that class. This approach works only if all applications running in a VM are using the same policy.

If multiple policies must be enforced, we need a way of distinguishing between versions of the API that enforce different policies. Naccio/JavaVM does this by statically renaming classes. The Java class file format makes renaming classes simple and efficient. All class names are given in the constant table found at the beginning of the class file. We replace class names of library files with the corresponding policy-enforcing library class name. The Naccio/JavaVM program transformer examines an application class to determine which classes it uses, and recursively examines those classes to determine all class dependencies. Classes that are not part of the Java API (that is, they are not described by the platform interface) are added to the classes to be transformed. The transformed classes are renamed and written into a new directory, preserving the original classes.

An alternative approach would be to select the API library classes at run-time. Wallach *et al.* describe how the Java ClassLoader could be modified to use namespace management to hide system classes or interpose implementations with extra security checking [Wallach97]. A similar approach could be used to select the appropriate policy-enforcing API class. The class loader would need to be written so that a request to load an API class would return a different policy-enforcing version of that class depending on the application calling the loader. This information is available by examining the class loader associated with the calling context. The static class renaming approach used by Naccio/JavaVM has the advantage that once the application has been modified it can be run repeatedly without further modification. Also, it means we are not tied to a particular Java environment. If applications that enforce different policies share objects that are instances of API classes, a type error will result. The problem of sharing objects between applications enforcing different policies is a complex one and is not addressed by the current design. Section 9.2 suggests some possible ways to support sharing objects across policies.

The other transformations that may be required in a policy description file are renaming native methods and inserting calls to initializers and terminators. For wrapped native methods, Naccio/JavaVM must replace the name of the method call with the wrapper name (e.g., w_write replaces write). JavaVM classes call methods by using a constant pool entry of type MethodRef that contains a reference to the class (an index to a ClassRef constant) and a reference to the name and type of the method (an index to a NameAndType constant). The NameAndType constant contains references to a name constant and a type descriptor, both represented by plain strings. To replace all calls to the method java.io.FileOutputStream.write with calls to java.io.FileOutputStream.w_write, the application transformer finds the constant pool entry that references this method and replaces its NameAndType constant with a new NameAndType constant that has the same type descriptor as the original but whose name identifies a (possibly new) string constant with value w_write. We cannot just replace the name in the old NameAndType constant, since constants that reference methods with the same name in other classes may reuse this constant. If there are no other references to the old NameAndType constant, it should be removed from the constant pool.

A special situation arises when an application class extends an API class with a wrapped native method. The situation is analogous to what is done for superclasses in the API by the policy compiler (as described in Section 5.4.1). If the subclass overrides the method, calls to the method for objects of the subclass type should call the subclass method. However, if those calls were rewritten to call the wrapped method (named w_*method*), then they call the superclass method instead and the incorrect behavior results. To ensure the correct behavior, the subclass must override w_*method* also. The application transformer inserts a new method named w_*method* into the subclass. Its implementation calls *method* with the original arguments.

If the policy requires initializers or terminators, the application transformer must modify the static main method of the class that will be used to start execution to call them. Java executions begin by calling the main method of the application class. This method should call each initializer at the beginning of execution, and each terminator before execution completes. This involves inserting instructions into the code body of the main method. If the policy requires violation codes, Naccio/JavaVM adds a new local variable of type naccio.library.ViolationCode and assigns it to the result of the newViolationCode static method. If the RSystem initializer is required, a call to it (passing the command line arguments as an array of strings) is inserted. After this, calls to the other initializers are appended. The violation code value is reset between each initializer call. Calling terminators is similar except it must be done immediately before each exit point. Exit points are the end of the code body and any return statements in the code body. The other way execution can terminate is by calling the java.lang.System.exit method. The policy compiler inserts calls to the necessary terminators in the wrapper for this method.

The other complication is that the main method may be called directly by the application. We must insure that the initializers and terminators are only called in the top-level main call. This is done by adding a static field named in_inner_call to the application class of type boolean that is initialized to false. Code inserted at the beginning of main assigns a new local variable to the value of in_inner_call and then sets in_inner_call to true. Code around the initializer and terminator calls tests this variable and skips the calls if it is true.

Java applets do not use a main method to control execution, but override the start and stop methods of the java.applet.Applet class. The start method is called to begin executing the applet, and the stop method is called when the applet should stop executing.[15] When an applet is transformed, the transformer treats the start and stop methods similarly to the main method of an application class. Initializers are called at the beginning of start and terminators are called before return points in stop.

If the policy requires no initializers or terminators, no wrapped native methods, and the CLASSPATH is used to select the policy-enforcing library classes, then there are no changes necessary to the application classes. It is not necessary to read or rewrite the application classes to enforce such policies. This means there is no load time cost associated with enforcing the policy other than setting the CLASSPATH appropriately. Many typical policies, including any policy that can be enforced using a JDK security manager, have this property.

6.1.2 Naccio/Win32

For Naccio/Win32, we need to alter the application executable so calls to API functions go to the appropriate wrapper DLL instead of the standard Windows DLL. There are two different ways a DLL can be attached to a process: listing the DLL in the import address table (IAT) in the executable image (called *implicit linking*), or calling the LoadLibrary API function to load the DLL at run-time (called *explicit linking*).

For implicitly linked DLLs, Naccio/Win32 can simply replace the DLL names in the IAT with those of the corresponding policy-enforcing DLL. Since the policy-enforcing DLL names differ

---

[15] It is up to the browser to decide the appropriate times to call these methods. Most browsers call start for an applet when the page containing it is visited, and call stop when the browser leaves the page containing the browser.

only in their three-letter extension, this replacement can be done by replacing bits in the IAT and does not require any code relocation. After the changes are made, the rewritten file must be rebound to ensure that function entry point addresses are updated to point to the policy-enforcing DLL. This can be done using the BindImage Win32 API function.

A wrapper for the LoadLibrary routine can be used to replace explicitly linked DLLs. Based on the name of the requested DLL, the LoadLibrary wrapper either loads the policy-enforcing version of the DLL or transforms the application DLL according to the policy. The policy description file includes a list of files used by the application transformer to determine how to handle a particular explicitly linked DLL.

6.1.3 Other Platforms

Although our experience is limited to the two prototype platforms, there are some general properties of the target platform and platform interface level that make it easy to interpose checking code. The system calls described by the platform interface must be easily distinguished from user code. In cases where they are linked dynamically, it should be fairly easy to change the library that is linked to interpose checking code. The approach used by Naccio/Win32 would work on any platform where the system API is linked dynamically and there is a way to replace which file is linked. Many modern platforms use some form of dynamic linking for system code. Some platforms, provide even better facilities for interposing checking code. For example, Solaris supports tracing of system calls using a user-defined function in a separate process. Janus (see Section 7.3.2) takes advantage of these features to interpose checking code on Solaris applications, and Naccio could readily be implemented on Solaris using a similar approach.

6.2 Guaranteeing Integrity

In a non-hostile environment, replacing the system libraries might be enough to enforce a safety policy on an execution. It is unlikely that a program would accidentally do something that circumvents or alters the checking done by the policy-enforcing library. Hostile attackers, however, may be motivated to take advantage of low-level manipulations to alter or avoid the policy checking. To guarantee the integrity of policy checking in these situations, Naccio implementations must ensure that it is not possible for hostile attackers to circumvent or alter the safety policy. They must ensure malicious applications cannot:

1. Manipulate resources in ways specified by the resource descriptions without going through a platform interface wrapper, for example, by jumping directly to API calls or using kernel traps.
2. Modify any checking code in resource implementations or platform interface wrappers. If the attacker can modify checking code, violations or resource operation calls can be removed to eliminate policy checking.
3. Modify the value of resource state or platform interface wrapper state. For example, if a malicious attacker could change the value of RFileSystem.bytes_written the LimitBytesWritten property could be circumvented. Being able to read this state is not considered a serious threat. Although clever attackers may be able to get some benefit from reading this information, it is not likely to be dangerous unless it is used in conjunction with some other vulnerability. Implementations that can prevent reading this state easily should do so, but it is not considered essential.

The measures taken to guarantee these properties lead to new properties that must be guaranteed. For instance, if any of the guarantees depend on static analysis or modification of the application code, Naccio must also ensure that the application cannot modify its own code during its execution.

What must be done to provide the necessary guarantees depends on the platform. Providing the necessary guarantees for Win32 is more challenging than for JavaVM, a simpler environment where security was considered in the design. In some cases, it may be necessary to disallow some harmless programs to provide the necessary guarantees. For instance, it is probably not feasible to distinguish between self-modifying code that circumvents safety checking and harmless self-modifying code so providing the necessary guarantees will involve disallowing programs that legitimately modify their own code.

6.2.1 Naccio/JavaVM

Naccio/JavaVM can take advantage of the properties ensured by the Java byte code verifier to limit the additional work that must be done. The Java byte code verifier [Yellin95] is designed to verify the low-level code safety properties required by Java. Before loading a class, the verifier performs data-flow analysis on the class implementation to verify that it is type safe, stack safe and that all control-flow instructions jump to valid locations. The class loader rejects classes that cannot be verified. All Java source code programs satisfy the low-level code safety properties, and it is up to compilers to generate code that can be verified by byte code verifiers. Naccio/JavaVM runs the byte code verifier before transforming a class to ensure it satisfies the standard low-level code safety properties.[16]

Hiding unwrapped methods

The Java byte code verifier is sufficient to guarantee that all jumps are either within a method, or method calls and returns, but not enough to guarantee that malicious programs cannot bypass checking code or manipulate state associated with a safety policy. We also need to ensure that the program cannot call the unwrapped versions of methods. The modified API class contains the o_*methods* that are copies of the original method as well as originally named methods that are unwrapped versions of native methods. The Java byte code verifier ensures the unwrapped o_*methods* are not called directly, since the application classes are verified using the original Java API libraries that do not define these methods. The program transformer replaces names of unwrapped versions of wrapped native methods with the name of the corresponding wrapped method (w_*method*) to ensure that unwrapped versions of native methods cannot be accessed directly by the application.

A malicious application could, however, attempt to access unwrapped versions of methods using the Java reflection classes. The class `java.lang.Class` provides methods that return the methods and constructors declared by a class. These can be used on any loaded classes, including the modified API classes. The methods are returned as objects of type `java.lang.reflect.Method`. The `invoke` method of this class can be used to call the returned method with chosen arguments. The implementation of `invoke` will throw an exception if the called method violates Java access rules, so reflection cannot be used to access private or protected methods inappropriately. Unwrapped routines, however, are declared with the same access modifier as the original routine since other API classes must be able to call them. If no efforts are taken to prevent it, an attacker could use

---

[16] In the prototype implementation, the verifier is run again on the transformed class. Security does not depend on this, but it is an easy way to detect bugs.

reflection to call the unwrapped version of a routine directly and thereby bypass all policy checking.

There are several feasible ways to prevent attackers from using reflection to call unwrapped routines. All involve using platform interface wrappers to restrict or alter the behavior of the relevant reflection methods. The simplest approach would be to disallow all the java.lang.Class methods that return method or constructor reflection objects. This would involve writing platform interface wrappers that issue violations for getMethods and the seven other similar methods that return method and constructor reflection objects. This would be an easy way to eliminate the threat, but it would also disallow useful programs that use reflection in a way that does not circumvent safety checking. A variation would instead use wrappers for the java.lang.reflect.Method.invoke and java.lang.reflect.Constructor.newInstance methods that issue violations before the method would be called. This would allow programs to view the unwrapped routines, but not allow any reflection object to be invoked. This provides the necessary protection but prevents less harmless programs that disallowing the reflection methods.

The next option is to write more complicated wrappers for the java.lang.Class methods that return method or constructor reflection objects. Instead of disallowing these methods completely, they would call the original method and examine the result. For non-API classes, the result should be returned. For API classes, the wrapper code checks if the result contains any unwrapped versions of wrapped routines (identifiable by their name starting with o_, by their name matching the name of another method starting with w_ for wrapped native methods, or by the dummy parameters added to constructors), these reflection objects would be removed from the result array before it is returned. This would allow programs to use reflection but prevent access to routines that would allow it to be used to circumvent safety checking. The risk is that the added complexity leads to more opportunities for bugs in the wrapper code that can be exploited by a dedicated attacker.

Naccio/JavaVM uses the first approach, using platform interface wrappers to disallow calls to the class reflection methods that reveal the methods and constructors. We believe that not enough Java programs use reflection non-maliciously to be worth the added risk of the more complicated solutions. Since reflection is a relatively new language feature, it remains to be seen if this solution would be adequate in an industrial implementation.

Hiding checking code and state

In addition to hiding the unwrapped versions of routines, Naccio/JavaVM must ensure that malicious attackers cannot manipulate state introduced by platform interface wrappers. State is implemented using instance and class fields added to the wrapped API classes, so Naccio/JavaVM must ensure programs cannot modify these fields. Since the state fields are declared private, application classes are not able to access these fields.

A similar situation arises with the generated resource classes. Programs must be prevented from either modifying resource state or calling resource methods. The most reasonable way to do this is to prevent application code from ever getting access to a resource object or class. As before, the Java byte code verifier prevents any explicit use of resource classes since they are not visible in the standard environment seen by the byte code verifier. The reflection methods can be wrapped to prevent access to resource implementation fields and routines. Another approach would be to use a platform interface wrapper to prevent java.lang.Class objects corresponding to resource classes from being created.

Dynamic class loading

The final thing Naccio/JavaVM must prevent applications from doing is dynamically loading classes that have not been transformed. If the application could load versions of the Java API classes that were not transformed to enforce the policy, routines from these classes could be called to manipulate resources without policy checking. Further, if the application could load classes from outside the Java API that were not transformed according to the policy, those classes could call API routines that manipulate resources without policy checking.

To prevent this, Naccio/JavaVM uses platform interface wrappers on the API routines (`java.lang.Class.forName` and several methods in `java.lang.ClassLoader`) that can be used to load a class dynamically. The simplest thing to do would be to prevent dynamic class loading completely by issuing a violation when these methods are called. This is likely to prevent too many harmless applications. Instead, the wrapper can load the appropriate transformed class instead. If the class to be loaded is a Java API class, the wrapper loads the renamed version of the class that enforces the policy. Otherwise, it needs to either locate a transformed version of the class or run the program transformer to create one. The other method that can be used to create a new class object is `java.lang.ClassLoader.defineClass`. This method creates a class object from an array of bytes representing the class file. Naccio/JavaVM could analyze the bytes to check if they enforce a policy, or transform the bytes directly. This was viewed as too complicated and risky to be worth supporting in the prototype implementation, and instead the wrapper for `defineClass` issues a violation for all calls.

6.2.2 Naccio/Win32[17]

Providing the necessary guarantees for Naccio/Win32 involves substantially more work than for JavaVM since Win32 provides none of the low-level code safety guarantees provided by the Java byte code verifier. Naccio/Win32 must perform protective transformations to provide the necessary guarantees. The prototype implementation does not implement these protective transformations. As a result, it could not be relied upon to provide code safety in a hostile environment. This section presents design ideas that could be used in an industrial implementation to provide the necessary low-level code safety guarantees. The program behaviors that must be constrained can be grouped into the three categories introduced earlier in this section: manipulating resources without going through platform interface wrappers, modifying code associated with policy checking, and modifying state associated with checking.

Protecting resource manipulations

There are several possible ways an attacker could attempt to circumvent platform interface wrappers. One vulnerability is that applications could manipulate resources without using Win32 API calls either by making direct kernel calls or by sending LPC messages to the Win32 subsystem. If the application can do either of these, it can manipulate constrained resources without any policy checking code being invoked. To prevent this, a static analysis detects all kernel and LPC calls in the program. Kernel calls are easily detected since they use special instructions to make a trap to the system kernel. LPC calls are more difficult to detect, but can be detected statically. Some calls can be determined to not manipulate a constrained resource. All other LPC calls are replaced with instructions that produce a violation. This leads to violations for some harmless programs, but it is uncommon for programs to use these techniques

---

[17] This section is based on [Twyman99, Chapter 5].

legitimately. Hence, it seems acceptable for Naccio/Win32 to disallow suspicious kernel traps and LPC calls completely. An ambitious implementation could attempt to write a platform interface for the kernel and LPC calls and insert calls to the necessary resource operations around the call. This would require substantial effort both in writing a platform interface at a lower level and transforming a program to insert the necessary code.

Another way an attacker could circumvent wrapper code is to jump to the unmodified DLL code directly. Since the policy-enforcing DLLs need to call the original API functions, the original DLLs must be loaded into the application's address space. Since Win32 binaries can use arbitrary values as addresses and jump to them, Naccio/Win32 must ensure that it is not possible to jump to an address that is in the original DLL or in the middle of the wrapper code. One technique for limiting the targets of jump instructions is software-based fault isolation (SFI) [Wahbe93]. SFI constrains the target address of jump instructions by inserting masking or checking instructions before the jump. The Naccio/Win32 design uses a variant of SFI to ensure that jumps in the application code can only jump within the application's code segment. In order to be able to make external calls to the wrapped DLL routines, stubs that make those jumps in a controlled way are added to the application code segment. Although SFI is well understood, actually implementing SFI on a Win32 platform involves a fair bit of complexity. Issues involved in adapting SFI to Naccio/Win32 are discussed in [Twyman99]. The prototype implementation does not implement SFI, so it unsuitable for use in adversarial situations.

Preventing code modifications

Naccio/Win32 must ensure that a malicious application cannot modify the checking code. Since we also depend on the static analysis and SFI transformations to prevent application code from making kernel or LPC calls or circumventing the wrapper code, the application must not be able to modify its own code or create new code. One approach would be to use SFI to prevent writes to the code segment to disallow any code modifications. The problem with this approach is doing SFI on every write is expensive and cumbersome.

Instead, we can take advantage of the virtual memory protection features provided by Windows NT and the Naccio wrapper mechanisms. The Win32 API provides functions for making regions of memory read-only or read/write. At the beginning of the initialization code, the code segments are marked read-only. This alone would offer no protection, since the application could call the Win32 API function to make the region read/write. However, we can use a platform interface wrapper to prevent this. The wrapper for the API function checks if the region that is being set to read/write is in the code segment. If it is, a violation is issued.

Protecting checking state

Naccio/Win32 must also protect state associated with checking. This includes state associated with resources and platform interface wrappers. We can protect this state from modification by application code by keeping it in a region of memory that is marked as read-only using the same technique as was use to prevent code modifications. The difference is the checking code may need to write to this state. To allow this, Naccio/Win32 must insert calls to the API routines to make the region writeable before the checking code and return it to the read-only state before returning to application code.

This works fine for single-threaded applications, but presents a vulnerability if the application has multiple threads. While the memory region is writeable to allow trusted checking code to modify the state, another thread that may be running malicious application code can modify the state without any violation being detected. This could happen either because the program is running on a multiprocessor machine, or because the operating system switches threads while the region is writeable.

In addition to the checking state, multiple threads also pose a threat to the local stack data for other threads. In particular, the local stack of a thread running checking code may contain temporary values that will be used in checking such as the absolute pathname corresponding to the file about to be opened. If a malicious thread is able to alter that stack data, it can disrupt the checking and prevent policy violations from being detected.

Protecting memory in the context of multiple threads is a difficult problem and no completely satisfactory solution is known. Twyman suggests some possible solutions [Twyman99]. Perhaps the most likely solution is to use SFI to protect memory writes. Since we can control where checking state is stored, using SFI to prevent writes in application code from modifying this state should be straightforward. Protecting local storage associated with checking code is more difficult since the regions that must be protected change throughout the execution. One solution would be to have a table in protected storage that records the regions that currently contain local checking storage. Checking code would write addresses into this table at the beginning of a routine, and remove them at the end. The inserted SFI instructions would need to check the write address against the regions in this table before allowing the write to proceed.

Chapter 7
Related Work

This chapter surveys work related to Naccio. The first three sections describe related work in code safety – Section 7.1 describes work in low-level code safety, Section 7.2 describes work in language-based code safety systems, and Section 7.3 describes work involving reference monitors. Section 7.4 describes other work involving program transformations. While most of this work was not directed towards security, the mechanisms used are similar enough to Naccio's to be worth including.

7.1 Low-Level Code Safety

Low-level code safety comprises the universal code safety required to isolate programs. It is primarily intended to protect memory references by prohibiting programs from reading, writing or jumping to certain segments in memory.

Early operating systems provided the necessary isolation using processes and virtual memory. The Multics operating system pioneered the use of virtual memory [Saltzer75, Denning80]. Virtual memory prevents processes from interfering with one another or the kernel by giving each process a separate view of the memory system. Instead of directly accessing physical addresses, a process uses virtual addresses that are mapped to physical addresses by a page table. The page table is in protected space and can only be modified by the kernel and the mapping is done by hardware on each memory reference, so there is no possibility of it being circumvented by a malicious program. The operating-system kernel is the only process that can see all of physical memory.

The problem with using processes for low-level code safety is that processes are expensive. A context-switch that may require substantial processor time is needed to switch between processes. Further, sharing data between different processes involves special mechanisms. As a result, researchers have sought to provide the same protections offered by hardware-level virtual memory by using software protections within a single process.

Verification systems

One way to provide code safety is to prove that the necessary properties are true about a program before it is allowed to run. One advantage of static verification is that after the properties have been verified, the code can run normally without any run-time overhead. The disadvantage is the properties that can be proved are limited by theorem proving technology and proving non-trivial properties typically involves substantial computation time. In theory, verification can be used to prove general code safety properties. In practice, it has been most successfully used to verify low-level code safety.

Java uses a byte-code verifier [Yellin95] to provide low-level security. Before loading a new class, the verifier performs data-flow analysis on the class implementation to verify that it is type safe and that all control-flow instructions jump to valid locations. Naccio/JavaVM relies on the Java byte-code verifier to guarantee low-level code safety. Although the verifications done are relatively simple, the byte-code verifier is still complex enough to contain bugs and the bugs are likely to be security vulnerabilities.

Proof-Carrying Code (PCC) [Necula96] is a more ambitions verification effort. PCC combines a program with a proof that the program satisfies certain properties. Before installing the program, a certifier verifies the proof. Proof generation may be complex and time-consuming, but verification is simple and efficient.

In theory, proof-carrying code techniques can be used to verify arbitrary properties about code. In practice, they are limited by automatic proof-generation technology, and only simple properties have been verified to date. [Necula98] presents a certifying compiler that takes source code in a type safe subset of C and generates optimized assembly language along with a proof that verifies its memory and type safety. Since all programs in the input language are guaranteed to have the desired properties, constructing the proof requires only that information present in the source code is not lost when it is compiled. Typed assembly language is used in a proof-carrying code system to verify type safety [Morrisett98]. A compiler can automatically generate type safety proofs for arbitrary programs in System F, a language supporting polymorphic types and first-class functions. Efficient Code Certification [Kozen98] seeks to verify low-level code safety using more compact and simpler certificates than those used in typed assembly language.

Proof-carrying code systems are limited since the producer of the code chooses the policy. The proof contains information needed to verify particular properties of the program, but provides no easy way to verify a different property. They may be useful for situations like operating system extensions when all that is required is memory and type safety, but are not able to offer sufficient flexibility to be useful in enforcing high-level safety policies. Another concern with proof-carrying code systems is the load-time overhead associated with verifying the proof.

The possibilities for combining verification with transformation-based run-time security are encouraging. Future hybrid systems will prove what they can about the original program, and then alter the program to make proving the additional properties easier.

Software Fault Isolation

Software Fault Isolation (SFI) [Wahbe93, TLLW96] enables a distrusted application to run in a shared address space without the possibility that it will interfere with memory outside its data segment. It works by altering memory access operations and jump addresses with bit masks to ensure that only the correct memory range is accessed. SFI was explained in more detail, along with the SFI-based mechanisms used by Naccio/Win32 in Section 6.2.2.

7.2 Language-Based Code Safety Systems

Static language-based approaches to code safety attempt to limit the damage a program may do by requiring that only programs written in a specific language be executed, and designing that language to have limited expressiveness. This can be done either by designing a new safe programming language or adding static checking to an existing language. The (unattainable) ideal safe programming language would be able to express all interesting safe programs and no unsafe programs. Actual safe programming languages either permit some unsafe programs to be expressed or prevent interesting safe programs from being expressed; most do both.

This work is relevant to Naccio, in that it presents an alternative way to safely execute code from untrustworthy sources. While language-based approaches has some appealing properties, the restrictions or demands they place on programmers limit their practical usefulness.

Type safety

A type safe programming language restricts a program's ability to convert values between different types. Providing type safety at compile time makes programs easier to understand and debug. Several type safe programming languages have been designed including Algol60 [Nauer63], CLU [Liskov81], ML [Milner90], Modula-3 [Nelson91] and Java [Sun96]. Type safety is generally a good trade off between increased reliability of programs and decreased language expressiveness, but it does limit the programs that can be written.[18]

Type safety can be used to provide the low-level code safety necessary to isolate programs by preventing programs from referencing invalid memory addresses. A language can provide this by checking types statically, preventing conversions between incompatible types, and limiting how particular types may be used. Combining this with forced initialization, automatic storage management and array bounds checking prevents a program from referencing arbitrary memory addresses and from manipulating memory in a way that does not correspond to its type. Type safe languages also limit what instructions a program may execute; all control flow is through language control structures and calls to well-defined procedure interfaces.

Restrictive programming languages

Other programming languages have been designed that provide more severe restrictions on programs. These languages are usually geared to a special purpose, and some are not Turing complete.

This approach was used in [Mogul87] to provide a safe way of allowing user code to implement packet filters that run in the kernel. A simple stack-based assembly language is used to encode a packet filter, and this is interpreted in the kernel. Since the packet filter language lacks any control flow operations, all programs are guaranteed to terminate.

PLAN [Hicks97] is a restrictive programming language designed for expressing programs that execute at the nodes of an active network. PLAN provides strong safety guarantees. PLAN programs are guaranteed to use a bounded amount of memory, processor and network bandwidth. PLAN does not support recursive function calls or unbounded iteration, hence, programs are guaranteed to terminate.

Both the packet filter language and PLAN place severe constraints on the programs that may be expressed. While they may be well suited for the particular application for which they were

---

[18] Here, by limiting the programs that can be written, we really mean limiting the possible implementations. Since all the type safe languages are Turing complete, any function that can be written in a non-type safe language can be written in all of the type safe languages. However, it may be more difficult to implement a particular program efficiently without the additional expressive power of a non-type safe language.

designed, they are not Turing complete languages and are not capable of expressing most useful programs.

Static checking

Another way to create a safer programming language is to add more static checking to an existing language. The most ambitious system using this approach to date is ESC [Detlefs96]. ESC attempts to prove at compile-time that certain errors (such as derefencing a null value, indexing an array out of bounds, or race conditions) will not occur. While ESC shows much promise as a debugging tool, it is unlikely that it could be used to enforce the kinds of high-level safety properties we are addressing. Many of these properties could not be checked statically since they depend on values that are not known at compile time (for example, a user enters a file name). Further, proving a property such as a constraint on the maximum number of bytes that may be written to a file is well beyond current and foreseeable automatic proving techniques.

Execution environment

Once a safe programming language is designed, a system can provide security only if the execution environment has some way of verifying that the program was created using the safe language.

The simplest solution is to use the source code in the safe programming language directly in the execution environment (PLAN uses this approach). The code can then be run in an interpreter, or compiled and executed. This approach has two main flaws:

- Performance – there is some performance penalty incurred by either having to interpret code or compile it every time it is executed. Just-in-time compilers offer some potential to reduce this performance cost.
- Code disclosure – most commercial software vendors view proprietary source code as the cornerstone or their business, and would be unwilling to develop programs for a platform that requires them to reveal their source code.

An alternative is to supply object code to the execution environment, but have some way for the execution environment to validate the object code. This can be done either by verifying that the object code was generated from a program in the safe language by a trusted compiler, or by verifying that the object code satisfies the safety properties of the safe programming language. SPIN and Java illustrate the two possibilities.

SPIN [Bershad95] uses extensions written in Modula-3 as a safe way of extending an operating system kernel. They suggest having a trusted compiler cryptographically sign the object files it produces. The execution environment validates an object file's signature before loading the code, to ensure that only unaltered code written in the safe programming language and compiled using the trusted compiler may be loaded. This approach depends on expensive cryptographic techniques, and prevents innovation or competition in producing compilers, since only the trusted compiler is able to sign code.

The other approach is for the execution environment to verify that the object code satisfies the safety properties guaranteed by the source language. In order to make the verification easier, it may be helpful for the compiler to include extra information in the object file. However, it is important that the verifier does not trust this information. The Java byte-code verifier and PCC (see Section 7.1) use this approach.

7.3 Reference Monitors

This section looks at other systems that use reference monitors to enforce security policies. The concept of a reference monitor originated in the early 1970s [Lampson71, Anderson72], and is described in Section 1.2. Here we look at a few reference monitor systems that are most closely related to Naccio. The diversity of systems represented illustrates the usefulness of reference monitors.

7.3.1 Java Security Manager

The only way a Java program may manipulate system resources is by calling provided Java API library functions or by calling native methods. Untrusted code is prevented from installing native methods, so security can be provided by placing limits on how the Java API routines are called. The API is implemented so that before an unsafe system call is executed, the relevant SecurityManager method is called. In theory, this guarantees that the reference monitor for a particular manipulation is always called before the manipulation is allowed. If the security policy disallows the call, a security exception is raised before the unsafe system call can be executed.

The SecurityManager is a Java class, so flexible security policies may be implemented. The scope and precision of policies, however, is limited by where the system libraries call SecurityManager check methods. The check methods are fixed by the API specification, and cannot be extended without changing the API specification and implementation.

A common paradigm in Java security policies is to use information on the call stack to determine what policy should be enforced. Every class and object at run-time has an associated class loader (a subclass of the java.lang.ClassLoader type) and the class loader reveals the source of the class. A typical SecurityManager policy uses this information to determine if the class was loaded locally or remotely, and enforces different constraints on different classes. The JDK 1.0 security model supported two types of code. Local code would run with no restrictions, and all remote code would run with severe restrictions imposed by a single SecurityManager implementation. JDK 1.1 extended this model to support signed applets that are treated as local code, but otherwise did not change the security model. To distinguish between types of code, authors of SecurityManagers must explicitly examine the ClassLoader stack.

JDK 1.2 (also marketed as Java 2 SDK) introduced a new security architecture that addressed many of the limitations of the earlier JDK versions [Gong97]. Unlike earlier JDK versions, where code was either trusted or untrusted, using JDK1.2 different code can run with different permissions. A system security policy defines a mapping between a protection domain and a set of access permissions granted to the code. Particular code is mapped to a protection domain based on its origin (URL location) and cryptographic signers.

JDK 1.2 also introduced mechanisms to make it easier to define a security policy in terms of setting permissions (as opposed to earlier releases where it was necessary to subclass the SecurityManager to change the policy). Permissions are defined as subclasses of the root java.securityPermission class. Typical permissions contain a target and an action. For example, the java.io.FilePermission class controls file system access. The target is a pathname (which may contain wildcards), and the action is one or more of read, write, execute and delete. Permission classes define a method implies that takes a Permission object and returns true if this permission implies the argument permission. Programmers can define new permissions associated with their application by creating a Permission subclass. When the permission should be checked, the code explicitly calls the security manager with a Permission object that represents the new permission.

This is useful extensibility, but it is up to the application programmers to define new permissions not the users or independent parties.

Instead of calling specific SecurityManager check methods, the JDK 1.2 uses the more general AccessController.checkPermission method that takes a Permission object. It will throw a security exception unless all classes on the call stack belong to protection domains that have been granted the requested permission. The normal semantics is that the permissions granted at a particular execution point are the intersection of the permissions granted by all protection domains in the call chain.

In exceptional cases, privileged code can call the AccessController.beginPrivileged to explicitly enable (and endPrivileged to disable) a particular privilege regardless of the protection domain of its callers. This is necessary to allow system API routines to manipulate resources even when they are called from an untrusted protection domain. Between the call to beginPrivileged and endPrivileged, all permission checks will ignore the permissions of callers further up the call stack and allow all permissions of the protection domain of the code that enabled privileges.

The method AccessController.checkPermission checks whether a particular permission is enabled checking. It can be implemented either by eagerly constructing the intersection of permissions when a code from a different protection domain is called, or by lazily looking up the execution stack when a permission needs to be checked. Sun's JDK 1.2 implementation uses the lazy evaluation approach [Gong98]. The other approach is used by the security-passing style [Wallach98]. Instead of searching the stack for protection domains, the stack information is encoded into a security context parameter that is passed as a parameter. This requires modifying Java classes to add and pass the extra parameter. The security-passing style has some advantages over the JDK 1.2 implementation since it is not tied to a particular JavaVM implementation and does not prevent certain compiler optimizations (such as inlining) that are not permitted using the stack searching approach. For typical programs, it is likely to perform worse than the lazy evaluation technique since passing explicit security contexts is more expensive then searching stacks when a permission needs to be checked [Wallach99].

Naccio avoids many of the complications associated with protection domains by making policy decisions at transformation time. There is no need to examine the execution stack to determine the protection domain of particular code, since that code has already been transformed to reflect the policy that applies to it. This eliminates the complexity and run-time overhead associated with stack inspection. It means, however, that certain policies that can be easily enforced using the JDK 1.2 mechanisms cannot be reasonably defined using Naccio. The relative expressiveness of Naccio in comparison to different code safety systems is discussed in Section 8.1.

7.3.2 Interposition Systems

Several systems have provided security by interposing checking code directly into the operating system. This can be done either by modifying the kernel or taking advantage of operating system features such as a tracing facility that support.

Although Naccio enforces policies at the application level, much of the work could also be applied to an interposition approach. The difference is that instead of modifying applications to use different policy-enforcing libraries the operating system library would be modified to call all standard resource operations. The resource operations would dispatch based on the policy in effect, which the operating system kernel can determine from the application running and a secure process-policy mapping. This has the advantage of eliminating the need to rely on low-level code safety of the application, assuming the operating system kernel is protected, as is the case in most modern operating systems. The other advantage is that no modification or analysis of applications is necessary; all that is required to enforce a policy on an execution is to select the desired policy. There are however, two substantial drawbacks to this approach. First, it requires access to the operating system kernel. Modifying an operating system kernel is usually a cumbersome and risky undertaking. Much of the modification, though, could be done automatically by the existing Naccio mechanisms. The other problem is performance. For every system call that can be constrained, it is necessary to check what policy is in effect and determine what if any checking code should be executed. This means that even unconstrained programs that are trusted completely will suffer substantial checking overhead.

Program-specific access controls

Several projects have sought to extend traditional operating systems with access controls that depend on the program executing. [Wichers90] suggests protecting a system from malicious programs by associating an access control list with each file that explicitly specifies which programs can modify the file. The access controls can be implemented through an extension to the UNIX kernel.

Cybermedia's Guard Dog [Cyber97a, Cyber97b] is a commercial product incorporating a similar idea to protect critical files in Windows. It includes a File Guardian that uses operating system hooks to monitor all access to critical files, and warns the user if a program not permitted to access the file does so. The user decides what programs are allowed to access particular files or communicate using the network.

TRON [Berman95] is a process-specific file protection system for the UNIX operating system. TRON allows users to create shells with specific access permissions that apply to all processes executed in the shell. A modified UNIX kernel enforces the permissions by placing wrappers around system calls.

Program-specific access controls have the advantage that safety checking is placed inside the operating system. This makes it harder for programs to circumvent the safety checking since the checking is conceptually close to the resource.[19] These systems have significant performance advantages over run-time approaches using a virtual machine, since untrusted programs are executed directly. The main disadvantage is lack of flexibility – checking is limited to a fixed set of predefined system calls. We could imagine support for checking a large number of system calls, but this has detrimental performance consequences. For each system call that is checked, all programs (both trusted and untrusted) incur the additional overhead of the safety check (although on some systems dynamically linking with a specialized library can minimize this cost). We have found no data that quantifies this performance cost well, but since it applies to trusted programs even a small penalty may be unacceptable to many users.

Janus

Janus is a system designed to limit the damage caused by untrusted helper applications used to process remote data [Goldberg96]. Non-malicious helper applications such as the PostScript viewer `ghostview` are complex enough that they are likely to have bugs that can be exploited by

---

[19] I suspect, however, that the program-specific systems (as opposed to process-specific systems like TRON) are vulnerable to attacks where a rogue program makes itself appear to have the identity of a trusted program.

data files constructed by malicious attackers. Janus limits what these helper applications can do by restricting their access to the operating system. Janus takes advantage of debugging features of the Solaris operating system that support tracing the system calls performed by an application. The tracing mechanism can be set to call a user-defined function in a separate process whenever a particular system call is issued. Since the checking code is in a separate process and the kernel provides the debugging features, no low-level code safety guarantees are needed to prevent the helper application from tampering with the checking code or data. This approach works well for Solaris, but could not be used on other operating systems that do not provide a similar tracing mechanism.

A policy is defined by a list of policy modules in a configuration file. A fixed set of policy modules is provided by the system. The configuration file controls the behavior of a policy module. For example, it can set parameters to the path module that control what directories can be read and written. Each module may return allow, deny or no comment on a particular system call. When different modules return conflicting responses, the later modules override earlier ones and no comment responses are ignored. If the last module that returns a response other than no comment returns deny, the system call is disallowed.

The module composition mechanism is similar to, but more general than, the policy composition mechanisms supported by Naccio. Where Naccio allows a property to be weakened by a permission using allow commands to override violations, Janus allows an unlimited number of modules to be combined with allow and deny responses overriding each other based on ordering. It is unclear whether the expressiveness advantages of this approach outweigh the added likelihood that a policy author will be confused and accidentally define the wrong policy.

Generic Software Wrappers

Generic Software Wrappers (GSW) is a technique designed to make off-the-shelf software more suitable for use in secure systems [Fraser99]. Prototype implementations have been developed for two Unix-based operating systems: Solaris 2.6 and FreeBSD 2.2. A wrapper support subsystem is implemented as a dynamic loadable kernel module, a feature provided by most UNIX systems. Wrappers run in kernel space so they are protected from application code and require no context-switch overhead.

GSW defines a policy by writing wrapper code in a C superset extended with some primitives useful in security checking. Wrappers are associated with system calls or system call groups introduced by annotations in the *characterized system call interface*. The characterized system call interface describes the system API. It is much less general and expressive than Naccio's platform interface, but motivated by the same desire to hide platform differences and allow safety policies to be expressed in a platform-independent manner. System calls are characterized by adding annotations to their return values, function names, and parameters. The annotations can be used to categorize functions, but not to precisely describe their behavior. For example, the annotations on the FreeBSD open system call indicate that it is a file operation that manipulates file descriptions, its return value is a file descriptor, and its first parameter is a null-terminated string representing a pathname. The library characterizations allow a wrapper to be attached to all system calls that deal with file descriptors. Within the code for that wrapper, however, it is not possible to determine how a file descriptor is being manipulated.

A Windows NT prototype implementation of GSW is currently under development [Spector99]. Since Windows NT does not provide support for dynamic loadable kernel modules, the standard architecture cannot be used. Instead, they use mechanisms similar to those used by Naccio/Win32. As with Naccio/Win32, it must enforce the necessary low-level code safety and they are attempting to do this by performing SFI transformations on running code [Feldman99]. If a Windows NT implementation of GSW were developed successfully, it would provide a useful platform to implement the low-level code safety necessary for Naccio/Win32.

7.3.3 Transformation-based Systems

A few systems have used program transformation approaches to code safety. These systems are similar to Naccio in their enforcement mechanisms, but differ in how policies are defined. In particular, all define policies at the level of concrete operating system calls or machine instructions.

SASI

Security Automata SFI Implementation (SASI) [Erlingsson99] is a generalization of SFI that can enforce a wide class of safety policies. SASI prototypes have been implemented for x86 assembly language output from the GNU gcc compiler and JavaVM code.

A policy is defined using a security automaton, similar to a finite state automaton. It consists of state and transitions where the input alphabet corresponds to events that a reference monitor would see. The input symbols correspond to program instructions – for the JavaVM version they are Java byte code instructions; for the x86 version they are x86 assembly instructions. This provides for unlimited precision, but makes it difficult to express even simple policies.

SASI converts a security automaton into code that is added to the program. New variables are added to represent the automaton states and code implementing the automaton is inserted between each program instruction. Unnecessary code is removed, and the necessary code is converted into machine code and inserted into the program executable. Unlike Naccio, the entire program must be analyzed and transformed instead of just replacing routine calls. This is necessary because policies are expressed at the level of individual instructions. In essence, an implementation of the security automaton defining the policy must be inserted before every instruction (fortunately, much of this can be optimized out for many instructions and policies).

Ariel

The Ariel project describes policies using a declarative language and enforces policies by inserting code in Java classes [Pandey98]. The transformations done by Ariel to enforce a policy are similar to those done by Naccio/JavaVM. Policies are described as access constraints that prevent the creation of objects or invocation of routines based on a predicate. Because of the declarative nature of policy descriptions, Ariel is unable to describe behavior-modifying policies that can be described using Naccio's mechanisms (such as the SoftSendLimit property described in Section 4.2.4). This, however, could be changed fairly easily by extending the policy language. Policies are described at the level of the Java API so they are not portable across platforms, and writing a policy that constrains writing would require placing constraints on all routines that may write to a file.

JRes

JRes is a resource management interface for JavaVM programs [Czajkowsik98]. It supports per-thread accounting for heap memory, CPU time and network usage. Limits can be placed on the amount of a particular resource a thread may consume, and callbacks are invoked when a limit is exceeded. In JRes, policies are described by application calls to methods that set fixed value limits on a predefined set of resources. Many policies that Naccio can enforce could not be defined using JRes because they depend on resource manipulations not constrained by JRes or they place more complex constraints on resource usage than a fixed limit (e.g., a rate or a function of other resource usage).

JRes is implemented by rewriting Java application classes to keep track of thread and resource information. To account for memory usage, JRes inserts code before every object or array allocation that calculates the size of the allocation and invokes a method that accounts for this memory usage. Accounting for CPU usage requires native code and a new thread that queries the operating system for CPU consumption.

The mechanisms used by JRes could be incorporated into Naccio/JavaVM with minor modifications. This would allow resources corresponding to CPU and heap memory usage to be defined, and policies to be defined and enforced that constrain these resources. Unfortunately, this would tie us to a particular JavaVM since JRes uses native methods and operating system calls to monitor CPU consumption.

7.4 Code Transformation Engines

Naccio depends on modifying program binaries to enforce a safety policy. Naccio/JavaVM uses an augmented version of the Java Object Instrumentation Environment (JOIE) toolkit to do the necessary transformations. The Naccio/Win32 prototype uses custom code to make simple binary transformations, but an industrial implementation would need a more substantial transformation engine to perform the transformations necessary to ensure low-level code safety.

The earliest known work on automatic program transformation for monitoring was the Informer measurement tool done at UC Berkeley in 1969 [Deutsch71]. Informer was developed to measure a time-sharing system by allowing user-written programs to be dynamically inserted as measurement routines. It would patch the operating system object code to call a measurement routine before an arbitrary selected execution point. More recent work has focused on providing tools that allow for more general program transformations, make the desired transformations easier to define, and support a range of complex platforms.

7.4.1 Java Transformation Tools

The Java byte code format is a popular target for code transformation engines since it is widely used, portable, well specified and far easier to deal with than most binary formats. Further, Java binary compatibility rules mean class files can be transformed in certain ways without breaking applications. Several tools for transforming Java class files have been produced including JOIE, Binary Component Adaptation, the Bytecode Instrumenting Tool, and Compaq JTrek. None of these tools were produced with security in mind, but rather improving performance and reusability of Java classes. Although any of these could have been used (with some modification) as the transformation engine for Naccio/JavaVM, we choose to use JOIE because at the time this work began it was the most mature and stable tool available, its source code was available, and it provided general enough interfaces to support most of the transformations needed by Naccio/JavaVM.

JOIE [Cohen98] is a toolkit for transforming Java classes. It is intended to be used to do load-time transformations by using a custom class loader that calls user-defined transformers. Naccio/JavaVM does not use the JOIE class loader, but uses classes in the JOIE toolkit to transform and rewrite classes independently from them being loaded into a JavaVM.

Binary Component Adaptation (BCA) [Keller98] is a tool for rewriting JavaVM code at load-time, designed to improve the reusability of Java components. Adaptations are expressed as changes that should be made to a class such as adding, renaming or replacing a method. A compiler converts the requested changes into a set of modification rules. When the class loader loads a class, it is modified according to the modification rules

The Bytecode Instrumenting Tool (BIT) [Lee97] is a tool for instrumenting JavaVM code, primarily directed at performance analysis. BIT supports insertion of code at key locations in a program (for example, method calls and basic blocks). BIT is not as general a transformation engine as BCA or JOIE, since transformations are limited to inserting code at points determined only by control flow.

Compaq JTrek [Compaq99] contains a class library that can be used to analyze and modify Java class files. It supports byte code transformations, intended to instrument classes with monitoring code. JTrek provides hooks for user-defined methods that are called when a routine is invoked or field is referenced and modifies Java classes to call those methods at the appropriate times.

7.4.2 Win32 Transformation Tools

Transformation tools for Win32 binaries are less readily available since there is substantial complexity involved in dealing with the Win32 binary format [Pietrek94]. One of the challenges in binary editing for Win32 platforms is code discovery. Unlike Java classes where the location of code and data is defined by the class format, distinguishing code and data in Win32 executables is complicated. Another problem is code relocation. If the length of code changes because of the program modifications, jump instructions and memory references must be adjusted to point to the modified location. This is particularly problematic for indirect jumps where the address is calculated and not known statically. Most binary editors rely on symbolic information that is part of the executable such as a debugging table identifying procedure entry points and data regions. Naccio/Win32 cannot depend on this information unless it is verified. There is no way to prevent an attacker from altering the symbolic information in a way that circumvents safety checking. All of these problems make transforming Win32 executables for security a challenging problem. Although it is believed to be possible, it would involve substantial effort and resources beyond what was available for the Naccio/Win32 prototype. The predominant Win32 architecture, Intel x86, poses additional problems because of the complexity of its instruction set. Supporting Alpha NT would be easier because of the simpler RISC instruction set, however a tiny fraction of Win32 users are using Alpha NT.

Several tools are available that would be a helpful starting point for an industrial implementation. OM [Srivastava92] is a tool for performing link-time modifications on Alpha binaries. It translates the program to a register transfer language and performs modifications on that representation before rewriting it as a binary. OM makes use of supplemental relocation information provided by the compiler in the binary. If it were used for code safety, this information would need to be verified or ignored. ATOM [Srivastava94] is a tool built on top of OM to simplify program instrumentation. It provides a set of APIs for instrumenting programs but does not support arbitrary modifications such as deleting instructions. ATOM has been used on the OSF/1, Digital UNIX and Windows NT operating systems. The Windows NT version of ATOM, Spike [Cohn97] provides binary instrumentation for Alpha Windows NT executables. In addition, it intercepts system calls using replacement DLLs to transparently substitute instrumented DLLs for their unmodified versions. A similar technique could be used by Naccio/Win32 to introduce wrapper code.

A few binary editing tools for x86 Win32 executables have been developed. Etch [Romer97] is a tool for rewriting x86 Win32 binaries. Etch analyses a Win32 binary to discover the code segments, and then cycles through each basic block instrumenting instructions. A (now-defunct) company, TracePoint, used OM technology to build tools that instrument Win32 binaries to do profiling and test coverage analysis [TracePoint97]. This work is believed to be continuing at Microsoft under the code name Vulcan [Srivastava98]. Neither Etch nor Vulcan is currently available for research purposes.

In addition to the single platform binary editing tools, a few projects have attempted to build general frameworks that can be used to edit binaries on different platforms. Executable Editing Library (EEL) [Larus95] is a C++ library for editing executables. EEL translates executables into a platform-independent register transfer language, allows transformations to be performed on the intermediate representation, and translates it back to a platform-dependent executable. EEL is intended to be portable across a wide range of instruction sets and binary formats, but so far has only been used with SPARC executables running under SunOS and Solaris and a partial implementation for RS/6000 AIX executables. It remains to be seen if this approach could work for a CISC architecture like x86.

The security system was adequate, but it did not foresee an armed robbery.

Italian Minister of Culture Walter Veltroni, explaining the theft of
two van Goghs and a Cézanne from Rome's National Gallery.

Chapter 8

Evaluation

This chapter evaluates the Naccio architecture and prototype implementations. We analyze how well the goals of security, versatility, ease of use, ease of implementation and efficiency set forth in Section 1.3 have been met by the Naccio design in general and our prototype implementations in particular.

8.1 Security

The most essential property of any security-related system is that it satisfies desired security requirements. For Naccio, this means that a specified policy is enforced correctly. There is no clear way to prove this in the positive, but any successful attack proves the negative. A formal analysis of the soundness of the Naccio design would increase our confidence, but is beyond the scope of this thesis. Instead, this section speculates on the security of the design and discusses likely vulnerabilities in the prototype implementations.

The smaller the part of the system security depends on, the more likely it can be implemented correctly or validated. Although Naccio's design is conceptually simple, the trusted computing base for Naccio is far larger than is desirable. In general, it comprises the program transformer, policy compiler, platform interface, and all system code below the level of the platform interface. For the Naccio/JavaVM prototype implementation, the trusted computing base comprises:

- The policy compiler. It must correctly parse the resource descriptions, resource use policy and platform interface. It must weave the checking code from the resource use policy into the resource descriptions. If optimizations are done to remove resource operations and platform interface wrappers, these optimizations must correctly determine that the removed modules do not do any useful checking. The code generated for the resource implementations must correctly implement the checking described by the resource use policy. Since a Java compiler is used to compile these resource implementations, that Java compiler is part of the trusted computing base also. The policy-enforcing library must correctly reflect the contents of the platform interface. This is perhaps the most complicated part of policy generation, and it is exceedingly unlikely that the prototype policy compiler does not have some bugs in the generation of wrappers. Further, it depends on the JOIE toolkit used as the transformation engine. Finally, the produced policy description file must accurately describe the transformations that must be done to enforce the policy on an application.

- The program transformer must correctly perform the transformations described in the policy description file. Naccio/JavaVM also relies on the Java byte code verifier to ensure low-level code safety properties. In addition, we rely on the wrappers for Java reflection and dynamic class loading correctly prohibiting applications from bypassing or tampering with the checking code. The prototype implementation keeps these wrappers simple (at the expense of disallowing some harmless programs) to increase the likelihood that they are correct.
- The platform interface must correctly describe the Java API in terms of the resource operations. The task is simplified somewhat by support for pass-through wrappers, but the platform interface must still correctly specify the behavior of several hundred API routines.
- The Java API implementation must not manipulate resources in ways different from those described in its documentation. Since the platform interface is written according the API documentation, if the Java API implementation produces different resource manipulations than described in its specification, an attacker will be able to exploit them to violate the safety policy without detection.

This is a very large trusted computing base, and it compares unfavorably with most other code safety systems. The trusted computing base for the JDK 1.1 security mechanisms comprises the byte code verifier, the Java API correctly calling `SecurityManager` check methods, and the `SecurityManager` correctly implementing that checking. It also depends on the Java compiler to correctly compile the API and `SecurityManager`, and the Java run-time to correctly distinguish between trusted (local or signed) and remote code and the `ClassLoader` only loading verified classes. This is certainly a larger trusted computing base than is desirable, and too large to be feasible to verify, but smaller than the Naccio/JavaVM trusted computing base. Systems like SASI [Erlingsson99] and proof-carrying code [Necula98] have smaller trusted computing bases than the JDK. Because they describe policies at the level of machine instructions, there is less processing needed (and hence, a smaller trusted computing base), to enforce or verify a policy.

The story for Naccio/Win32 is similar. Its security depends heavily on correct implementation of the protective transformations necessary for low-level code safety. Implementing SFI is notoriously difficult for a platform as complex as Intel x86 and no satisfactory implementation that deals with arbitrary Intel x86 executables is known. Further, the Win32 API is large and complex. The prototype implementation only defines a partial platform interface; correctly defining a complete one would constitute a major undertaking.

One way to deal with a large trusted computing base is to identify and verify the most vulnerable pieces. The platform interface is the most likely candidate for verification. Section 9.1 discusses some possible ways of increasing confidence that various parts of a Naccio implementation are correct. The other way is to change the design or implementation to shrink the trusted computing base. One way to do this would be to move more of the checking into the operating system. On platforms that support extensible kernel modules (such as Solaris and FreeBSD), this could be done without any need to modify the kernel. This would eliminate any reliance on low-level code safety, other than trusting the operating system mechanisms that protect the kernel. The trusted computing base would then only be the policy compiler that generates the checking code from the policy and the platform interface that describes the kernel calls. One way to simplify and reduce the size of this code would be to remove all the optimizations. This would incur a significant performance penalty, but would be acceptable in situations where greater security assurance is more important. If Naccio/JavaVM enforced policies at the level of native methods instead of the Java API, it would eliminate much of the trusted computing base since it would only rely on hooks into the native methods and the generated checking code itself. It would remove reliance on the Java API implementation, other than the implementation of native methods. The platform interface would be smaller and simpler, since it describes only security-relevant native methods. The drawback is it would tie Naccio/JavaVM closely to a single execution platform. It would also be more difficult to write extended safety policies, since the platform interface must be expressed at the level of machine instructions.

Naccio's large trusted computing base is one of the prices we pay for abstract policy definition mechanisms. The further away the policy definition is from the execution platform, the more work that must be done to enforce the policy. While the tradeoff between increased trusted computing base size (and the resulting reduction of confidence in the security mechanisms) and the ability to efficiently enforce a wide class of useful policies may be acceptable for low and medium security environments, it is not acceptable in security-critical environments. For security-critical environments, Naccio may be usefully combined with simpler enforcement mechanisms with better assurance that enforce the most important properties in such situations.

In addition to the sheer size of the trusted computing base, some aspects of the prototype implementations are of particular concern. Naccio/JavaVM supports pass-through wrappers to make writing the platform interface easier. This greatly reduces the size of the platform interface needed for the Java API. On the other hand, it increases the complexity of the policy compiler. Handling pass-through wrapper semantics is the most unwieldy part of the policy compiler implementation and the most likely part to contain bugs that are manifest as security vulnerabilities. Nevertheless, we believe the benefits of supporting pass-through wrappers in reducing the size of the platform interface outweigh these risks. Another possible vulnerability of the prototype Naccio implementations results from the optimizations done by the policy compiler to remove unnecessary resource operations and platform interface wrappers. Bugs in these optimizations can lead to wrappers that do meaningful checking being incorrectly removed and as a result produce a policy-enforcing library that does not detect violations of the policy. We believe the analysis is simple enough to implement correctly so that the run-time performance benefits obtained by removing unnecessary wrapper more than outweigh the added risks associated with bugs in the optimizer code.

The Win32 platform presents some additional vulnerabilities not faced on the JavaVM platform. Ensuring low-level code safety is much more difficult, and is not attempted by the prototype implementation. We believe it is possible to implement SFI correctly on Win32 Intel x86 executables, although it remains to be seen if this is true. Multiple threads pose another problem, and there is some doubt as to whether or not a satisfactory solution to protecting wrappers and resource state in the presence of multiple threads can be found.

Even if a Naccio implementation is correct, attackers can still exploit poor policy choices. Since all constraints imposed by Naccio are discretionary, it is up to users and system administrators to determine a suitable policy for their environment. Actually deciding what is an appropriate policy for different environments is beyond the scope of this thesis, but it is important that precise enough policies can be expressed and that it is easy enough to define and understand policies that it is likely a policy really means what its author intended. We believe Naccio offers some advantages over the alternatives because of the way policies are described in terms of abstract resource manipulations. The next two sections discuss this.

8.2 Versatility

This section considers how well Naccio encompasses the range of useful safety policies. We consider the issue in general, and then specifically for the standard policies and extended policies supported by the prototype implementations.

8.2.1 Theoretical Limitations

The policies Naccio can enforce encompass all safety properties that can be expressed in terms of manipulations visible at the platform interface level. With a suitable platform interface, all resource manipulations are visible and Naccio can define and enforce all policies in Class EM (see Section 3.3). Naccio cannot enforce liveness properties or policies that depend on knowledge of all possible executions.

Liveness properties depend on knowing something will happen in the future. For example, a policy that requires that all open files must eventually be closed is a liveness property. Although Naccio cannot strictly enforce liveness properties, most useful liveness properties can be approximated. For example, we could modify the file close policy to require that all open files must be closed before the application terminates. Naccio could define this policy by adding a state block that maintains a set of the currently open files. Code associated with the open file resource operations would add files to the open set, and calls to the close operation would remove them. Checking code associated with the file system terminator could either issue a violation if the open files set were non-empty before execution is about to terminate. Approximations of liveness properties may be slightly awkward to express, but Naccio can approximate many of the liveness properties that are useful for security.

Policies that depend on knowledge of all possible executions cannot be enforced without static analysis of the program text. Most properties in this category deal with information flow. Knowing whether a particular execution reveals information about some object requires determining if the visible output of this execution is distinguishable from other executions where the value of the object is different. Since runtime monitors on a single execution cannot reveal if this is the case, Naccio cannot be used to enforce fine-grain information flow. Naccio can be used to enforce coarse information flow policies that prohibit any remotely visible behavior after a sensitive object is touched. For example, a policy could be defined that prohibits all network use after any sensitive file has been read. In situations where fine grain information flow policies are essential, it would be necessary to combine Naccio with a static analysis tool that enforces the fine grain information flow policy.

8.2.2 Policy Expressiveness

Although in theory a platform interface can be created that makes all resource manipulations visible to the policy author, in practice it is not usually practical to do so. Both prototype implementations use platform interfaces at the level of a system API. This limits the policies that can be enforced to those expressible in terms of events visible through system API calls.

First, we consider the class of standard policies since those policies that can be defined using the standard resources represent the class of policies that can be easily defined. In addition, standard policies are portable across Naccio implementations. Porting an extended policy requires altering the platform interface on each new platform. Hence, it is important that most common policies can be expressed as standard safety policies. Standard policies can be used to express access control policies on any of the standard resources including files, network connections, windows, and threads. In addition to the standard static access control policies, policies that constrain access control dynamically based on the history of all resource accesses made by the execution can be written by using state blocks. This covers most traditional access control security policies.

With extended policies, the class of expressible policies expands to include constraining and modifying all behavior visible at the level of the platform interface. For the prototype implementations with platform interfaces at the level of the system API, this means all resource manipulation done through the system API can be constrained. By using state blocks, Naccio can define any policy that depends on the history of all system calls made by the execution. This is a large class of policies, but there are still limitations on what policies can be expressed. In addition to the theoretical limitations discussed earlier, the expressible policies are limited by what is visible to the platform interface. Some resources are manipulated without using system calls, in particular memory and the CPU. Naccio implementations cannot place any constraints on manipulating resources that are not visible at the level of the platform interface.

Other resources are visible at the application level, but do not correspond to any system resource. For example, a library may maintain a database in local storage and provide routines for manipulating that database. Since these routines are not part of the system API calls, there is no way to use Naccio to enforce a policy that constrains how the database may be manipulated. Eventually, the database manipulations may lead to a file modification or network transmission that can be constrained by Naccio. However, it is likely to be difficult to define a database access policy in terms of file operations, since the mapping between file segments and database entries is often complex and dynamic. It is possible, however, to extend the platform interface to include the database classes and to define new resources that correspond to manipulating the database.

Comparison to JDK

Naccio/JavaVM can mimic any JDK policy since we can write a Naccio policy that makes the same calls to the security manager check methods at the same execution points and with the same parameters as the Java API does (the MimicJDK policy introduced in Section 8.5.1 does this). Although this clearly duplicates a JDK policy, it is not entirely satisfying since it depends on hooks that allow policies to call Java methods. The policy is not portable because it relies on the Java SecurityManager code.

To define a portable version of a JDK policy, the checking code needs to be moved directly into the safety properties and translated into the generic property language. One problem that needs to be addressed is how to deal with JDK code that distinguishes between privileged and unprivileged code. The JDK 1.1 security manager often depends on examining the ClassLoader to determine if code is part of the system and should be considered privileged. JDK 1.2 uses stack inspection to provide a more general way to enable privileges through a call sequence.

Duplicating stack inspection requires access to more run-time state than is visible in a Naccio policy. The stack is not visible from a safety policy, so there is no way to define a policy that treats resource manipulations differently depending on what is on the call stack when they occur. Naccio can, however, mimic most of the useful aspects of stack inspection. Further, we argue that many policies defined in terms of different privileges supported by stack inspection are better expressed as more precise policies not depending on different privileges.

In JDK 1.1 and earlier, stack inspection was limited to distinguishing system and application code based on the ClassLoader. Naccio makes the same distinction at the platform interface boundary. Code within the Java API that is described by the platform interface is trusted. The wrapper describes its behavior and the implementation code runs with no additional safety checking. To define a policy that allows system code to manipulate resources in ways not permitted by application code, all that is necessary is to write a wrapper for the relevant API routine that does not call the resource operations corresponding to its actual manipulations.

JDK 1.2 supports a richer model where stack inspection distinguishes arbitrary classes of code. This is primarily motivated by the desire to support multi-layered applications where different classes have different trust levels and capabilities. Given that Naccio cannot duplicate stack inspection behavior exactly, we need to consider what is lost in terms of expressiveness as a result. Stack inspection was motivated by the desire to allow trusted code to perform actions that untrusted code is not permitted to, even when that system code is called by the untrusted code. A common example is the font loading code in the AWT. This needs to open and read a file, but is viewed differently from an application attempting to open and read a file directly. An example that takes advantage of JDK 1.2 capabilities would be an untrusted application that calls a third-party library that calls system code. The system code is privileged, and the third-party library has some privileges not afforded to the application but does not have all the privileges available to the system code.

This allows certain policies not expressible using Naccio to be defined, but it is questionable whether or not these kinds of policies are desirable. Policies expressed in terms of varying privileges do not correspond well to anything a user understands. Users have no notion of stack frames and make no distinction between system code and application code. It is awkward to describe to a typical user a policy that allows system code to access the file system but does not allow application code to do so. This policy might be useful if we want to allow the AWT font loading code to read a local file but prevent application code from directly accessing the any files. The assumption is that it is okay for system code to do this, since the system code is trusted and only limited information about files is made available to the untrusted code as a result. This seems contrary to the user's understanding of the policy. It would be better to define a more precise policy that constrains only the behavior of the program but make no distinction between what code is directly causing that behavior. For example, a better policy would disallow access to files except allow reading a limited number of files in the system fonts directory. This policy could be easily defined using Naccio by weakening a no writing policy with the standard JDK allowances that permit reading the font files. We believe most useful policies that depend on varying privileges can be better defined as precise policies defined in terms of program behavior.

There is a wide class of policies enforceable by Naccio but not enforceable by the JDK security mechanisms. This includes policies that depend on resource manipulations that do not correspond to security manager checks and are not visible to JDK policies. In order to enforce these types of policies using JDK mechanisms, additional check methods would need to be added to the `SecurityManager` and API implementations would need to be altered to call those check methods at the appropriate execution points. For example, current JDK mechanisms cannot support policies that constrain file activity once a file has been opened. This means both that there is no way to revoke read and write access once it has been granted by the original open call, and that there is no way to constrain the amount or content of data read from or written to an open file. To extend the JDK to support this class of policies, one would need to add new security manager check methods that correspond to reading and writing to files. All the Java API classes that read from or write to files would need to be modified to call the new check methods at the appropriate time. Even supposing one did have access to change the Java API in this way, the overhead of calling the new check methods is suffered for every Java program that runs with a policy enforced, even if that policy places no constraints on reading and writing to files. This would be unacceptable in many environments, since the overhead associated with security checking for a commonly called routine like writing a byte to a file would be substantial. Naccio avoids this problem by inserting the platform interface wrappers only when they do useful work. Hence, there is no overhead suffered unless the policy constrains a particular resource manipulation.

Examples

One way to get a better handle on whether Naccio can successfully express useful policies is to consider whether policies that protect against particular kinds of hostile applets can be written. One well-known collection of hostile applets is Mark D. LaDue's collection of hostile applets [LaDue96, LaDue99]. Here we consider how effective Naccio would be in protecting against each of these applets. The effectiveness of Naccio's policy definition mechanisms is judged on the basis of whether a policy can be written to prohibit the attack, how easy it is to write the policy, and how precisely it excludes the hostile behavior.

- The NoisyBear applet displays a clock and makes an annoying sound. Even after you leave the page, the sound continues. This behavior could easily result from an accidental programming error, and takes advantage of browsers allowing applet threads to continue even after the browser has left the page containing the applet. The simplest policy that would prevent this particular attack would be to always disallow playing audio files. This would disallow some potentially useful applets, however.

  The more general problem revealed by this applet, however, is that threads are allowed to continue after the applet stop method has been called. The browser calls the applet stop method when it leaves the page containing the applet, but there are no requirements that the applet stop method actually terminates all applet threads. Using Naccio, we can impose a policy that requires that no applet threads are running at the end of the stop method. The program transformer modifies the applet to call terminators (including RSystem.terminate) at return points of the applet stop method. A useful policy would keep track of all threads created by the applet using a state block, and then check that all threads have been stopped when RSystem.terminate is called. Occasionally, a useful applet may need to keep threads running after the containing page has been left by the browser. It seems reasonable to require user approval before allowing this.[20] The JDK approach could not be used to enforce such a policy, since there is no check method associated with stopping an applet.

  The best general solution to these kinds of attacks, however, is at the browser level. A security-conscious browser should allow the user to see what applet threads are running and which URL was responsible for their creation, and to selectively kill annoying or suspicious threads.

- The Consume, Wasteful, HostileThreads and TripleThreat applets are all denial-of-service attacks that consume most available CPU and memory. They work by creating a new thread, setting its priority to MAX_PRIORITY, and doing lot of useless processing. A policy that disallows increasing a thread's priority would solve part of the problem since a normal priority thread will not prevent other threads from acquiring the CPU. This policy can be easily defined using Naccio by issuing a violation in the standard resource operation associated with setting a thread's priority if the requested priority is too high. A less obtrusive policy would not issue a violation, but instead skip the system call that sets the priority. Defining this policy requires changing the platform interface so the original system call can be skipped.

---

[20] Supporting this well would require changing the terminators, so that different resource operations correspond to stopping the applet and termination of the last thread associated with the applet. This could be done by transforming applet code so each thread checks if it is the last thread running before completion, but would perhaps be better done by the containing application.

This would lessen the effects of the denial-of-service attacks, but would not prevent the eventual consumption of resources. To do this we need a policy that restricts the actual resource use. Attack applets use different kinds of resources in different ways. Some create a large number of threads or windows. Naccio policies can easily place limits on the number of threads or windows created, and it seems sensible to import such a policy on untrusted applets. If the resource use is done through processing and memory allocation, however, Naccio/JavaVM is not able to constrain the resource consumption. The limits on thread creation and setting thread priorities, would significantly reduce the amount of the CPU resource that could be consumed, but Naccio/JavaVM cannot enforce a policy that places limits on memory and CPU usage. Since using memory and the CPU does not correspond to a system call, these are not visible to the platform interface so we cannot write a policy that constrains them using Naccio/JavaVM. Although it is possible to extend Naccio/JavaVM to support these resources (see Section 9.2), it is more practical to constrain memory and CPU usage in the run-time environment.

- AppletKiller is a hostile applet that shuts down all other applet threads. It recursively loops through threads in a thread group, and their parents. Naccio policies can easily place restrictions on killing threads by writing a resource use policy that attaches checking code to the RSystemThreads resource operations. Perhaps the most reasonable thing to do would be to disallow any access to the applet's parent thread. A Naccio policy can do this by attaching checking code to the resource operation associated with getting a thread or thread group's parent. The difficulty is determining if the requested thread is the original applet thread (in which case the call reveals information about threads outside the applet), or a thread created by the applet (in which case the call should be allowed). To do this, we need a state block that keeps track of how a thread was created.
- Forger sends forged email by opening a network connection back to the originating host that uses the send mail port (25). A simple policy that would prevent this would prohibit all connections to port 25, or more generally prohibit explicit connections to any questionable port. This policy could also be written easily using a JDK security manager. The problem is with the default settings and policy interface on most browsers, not the available JDK security mechanisms.

To summarize, all of the applets in the hostile applets collection can be mitigated using standard safety policies. Only the denial-of-service applets that consume memory and the CPU but not some constrainable resource cannot be prohibited by a reasonable policy. The policies can be expressed precisely enough that the hostile behavior can be prevented without also preventing many non-hostile applets.

It is not clear how well the hostile applets collection corresponds to the real attacks browsers are likely to face. In fact, there have been few reports of malicious attacks exploiting Java[21]. Nearly all the media reports of Java vulnerabilities result from academic research rather than discovery of an actual malicious attack. Despite Java's security vulnerabilities, it is far easier for a malicious hacker to cause damage in other ways and most attacks exploit Windows executables or (more recently) macros for Word and similar programs. Fortunately, a Windows implementation of Naccio could be used to prevent many of these attacks. Java may become a more popular target for attackers as users become more security conscious and resist running

---

[21] Symantec's database of about 40,000 viruses and Trojan horses [Synamtec99] contains only two Java viruses (strangebrew and beanhive). Both depend on running in an environment where file access is not constrained. Trojan horses should be more common, but there are none reported in Symantec's database.

untrusted programs without code safety. However, for the near future, it is likely that buggy programs are a more serious threat to users with Java-enabled browsers than are malicious attacks.

8.3 Ease of Use

A main goal of Naccio is to make it easier to write, modify and understand policies than it is with other systems. This is a subjective question, but can be considered by looking at how much knowledge and code is required for different kinds of policies. Our experience with actual users is limited to that of the author and Andrew Twyman's experience developing policies the constrain network use before he began to develop Naccio/Win32. He was able to write new policies after looking at a few examples and had no problems defining the desired policies using Naccio.

Many policies can be written by combining and setting parameters of predefined properties. Users can construct these properties without any knowledge of resource descriptions or how policies are defined. If a sufficiently comprehensive property library is included with a Naccio implementation, it should be possible for most users and system administrators to construct many of the policies they need using predefined properties.

More sophistication is required to write a new safety property. We hope that moderately sophisticated computer users without substantial programming experience will be able to understand and write standard safety policies. To do so requires being able to understand the concept that program manipulations are characterized by resource operations, and that attaching checking code to resource operations constrains those manipulations.

The simplest policies are expressed as checking code that attaches a violation to a resource operation. For example, consider what must be done to write a policy that prohibits altering or creating files but allows reading. To define this policy using Naccio, a policy author must determine that file writing corresponds to the RFileSystem resource, examine the RFileSystem resource description to deduce that the modifyFile group corresponds to altering or creating files. The policy can be defined by attaching a violation to RFileSystem.modifyFile. Writing the same policy as a JDK SecurityManager involves creating a new SecurityManager subclass and overriding the checkWrite and checkDelete methods to throw an exception.[22] In fact, the default SecurityManager disallows everything, so the policy author either needs to subclass a different SecurityManager, or needs to override every other check method with an empty body. This involves a fair bit more programming knowledge than writing the Naccio policy (understanding subclassing and exceptions), but perhaps less effort for someone who already knows Java. Most Java-enabled browsers do not support user-written security managers, but instead provide a graphical interface for setting security parameters. The policy configuration dialog boxes for Internet Explorer 5.0 cannot be used to define a no writing policy. The only choice is to either enable or disable access (both reading and writing) to all files or to require a user prompt to approve all file access. While the interface for selecting policies is simple enough for naïve users to understand, it places severe limits on the range and precision of policies that can be defined.

---

[22] The java.io.File.rename method calls checkWrite on both its arguments; java.io.File.delete calls checkDelete.

The next level of complexity in writing policies is writing policies that maintain state, such as `LimitBytesWritten` shown in Figure 6. Writing state-based policies involves more programming, but Naccio's mechanisms make it easier to write these policies than the alternatives. In addition, a library of common state blocks covers the state needed for many policies. It will often be possible to express a new policy using pre-defined state.

The fact that defining a Naccio policy requires writing code makes it inaccessible to the majority of computer users. The subset of users who might be willing to consider writing a safety policy is probably similar to the class of users who write their own spreadsheet or Word macros. To make Naccio accessible to a wider class of users would require a graphical, parameter-based interface to policies. Such an interface tool could be created, but it is beyond the scope of this thesis.

8.4 Ease of Implementation

This section considers the amount of effort required to produce a new Naccio implementation by examining the effort required to produce the two prototype implementations. We report on how much work was required to produce Naccio/JavaVM, the first Naccio implementation; and how much additional work was needed to produce an implementation of Naccio for another platform, in this case, Naccio/Win32. Many of the lessons learned from these efforts would reduce the amount of time needed to produce a new Naccio implementation. Further, because of the design of the Naccio architecture, much of the code from the policy compiler can be reused on implementations of Naccio for different platforms.

Implementation of the Naccio/JavaVM prototype began in May 1998. Before this a preliminary prototype had been produced that transformed ANSI C source code according to fixed rules as a proof-of-concept[23], and the Naccio architecture had been designed and described in a thesis proposal. It took about four weeks to build a basic Naccio/JavaVM system that could be used to enforce simple policies on test programs. The main difference between the original prototype and the implementation described in this thesis is that instead of modifying the Java API class files to produce the policy-enforcing wrapper classes, the original implementation generated subclasses as Java source code and ran the Java compiler to produce a class file. The program transformer replaced calls to constructors for wrapped classes with calls to the corresponding constructor in the generated subclass. Generating Java source code subclasses is much easier than rewriting byte codes as is done by the final implementation, but had some significant drawbacks. It made it awkward to constrain final methods since they could not be overridden in the subclass. We could work around this problem using a similar technique as is done to handle wrapped native methods in the current implementation – rename the wrapped final methods and replace names in program transformation. Dealing with constructors posed another problem, since Java compilers require the call to the superclass construction be the first statement in the constructor body. This meant checking code could not be inserted before the original constructor call. More fundamentally, the subclassing approach suffered a non-negligible run-time overhead associated with the additional virtual method calls that would be suffered even for methods not constrained by the safety policy.

As a result, the implementation was changed to rewrite the Java class files directly. Implementing the class modifying code took a few weeks. Much of this time was spent learning the intricacies of the Java byte code format and understanding and modifying JOIE (see Section

---

[23] This was done because the author had access to and familiarity with a tool that deals with ANSI C code [Evans96], from which a simple program transformer could be constructed with little effort.

7.4.1) to support the necessary changes. The Naccio/JavaVM prototype including the policy compiler and program transformer as described in this thesis is implemented in about 40,000 lines of Java code, some of which were generated automatically by a parser generator. Most of the code is for the policy compiler (although the actual division is not obvious since they share objects and code) and is reusable for Naccio implementations for other platforms. The core of the program transformer is about 1500 lines.

The Java API platform interface was developed in conjunction with the Naccio/JavaVM implementation and test policies. Although pass-through wrappers were not part of the original design, the need for them became apparent early in the process of developing the platform interface. Support for pass-through wrappers greatly reduced the amount of work needed to write the Java API platform interface.

The second Naccio implementation was Naccio/Win32, built by Andrew Twyman starting in January 1999. Building Naccio/Win32 involved changing the policy compiler back end to produce C code for resource implementations instead of Java classes, and creating new tools that produced the platform interface linker file and modify the executables import table. Except for developing a new back-end, the rest of the policy compiler was reused without any changes. Converting the back end to produce C code instead of Java took a little over a week, and did not require a deep understanding of the rest of the policy compiler. Since the Naccio/Win32 prototype did not implement the protective transformations necessary for low-level code safety, the amount of work needed to implement the program transformer was limited to replacing DLL names in the import table. This was accomplished in a few days; almost all the effort was in understanding the Windows executable format. In addition, we did not produce a complete platform interface for Win32. Because of the size and complexity of the Win32 API, construction of a complete platform interface would likely take a skilled developer several months. The prototype platform interface used by Naccio/Win32 that only covered a subset of file manipulation calls took about two weeks to write and debug.

Producing the next Naccio implementation should involve less work that was necessary to produce the first two. The Java and C back-ends to the policy compiler should provide a good starting point for producing resource implementations on most platforms. For example, either back-end could be fairly easily adapted to produce resource implementations suitable for a Linux implementation of Naccio. There are two approaches to generating the policy-enforcing library exhibited by the prototype implementations – Naccio/JavaVM modifies the object code directly, while Naccio/Win32 generates separate wrapper code that performs the policy checking and then calls the original routine. The Naccio/JavaVM approach does not transfer easily to a new platform since modifying object code is likely to be highly platform-specific. The Naccio/Win32 approach is likely to be reusable on other platforms. To implement similar wrappers for a Linux implementation, it would be necessary to write a suitable platform interface and produce the appropriate linking information, but otherwise most of the Naccio/Win32 implementation could be reused.

The standard resource library evolved during the course of developing Naccio/JavaVM and Naccio/Win32. Some changes to the resource descriptions were a direct result of experience building Naccio/Win32. For example, the operations dealing with file observations were inadequate to precisely reflect all the different file properties that may be observed using the Win32 API. New operations were added to the `RFileSystem` resource corresponding to operations like observing the creation time of a file. It is likely that the standard resource library would change slightly as a result of producing a Naccio implementation for another platform. We expect it would converge fairly quickly, though, and after one or two more platforms there would be no need to change the standard resource library to support a new platform.

Finally, we consider what would be necessary to produce industrial quality implementations of Naccio. Naccio/JavaVM is close to what an industrial implementation would be, except for lacking the validation necessary to provide good security assurances. The amount of effort required to do this is substantial, but similar to what would be required for any code safety system. Producing an industrial version of Naccio/Win32 would involve implementing the protective transformations necessary for low-level code safety. While the work required is substantial because of the difficulties in implementing software fault isolation on the x86 platform, almost all of it is the same as would be required for any code safety system that runs x86 executables directly. If a satisfactory implementation of software fault isolation were available, it could be adapted to support Naccio with only minor changes necessary to protect the state associated with safety checking. The other major task necessary to produce an industrial quality Naccio/Win32 implementation is producing a platform interface for the Win32 API. This would involve substantial effort because of the size and complexity of the Win32 API.

8.5 Efficiency

The performance of a code safety system is important since a system that incurs a significant performance penalty will not be acceptable except in a security-critical environment. With Naccio, the costs of enforcing a policy are divided into three phases. First, the policy compiler is run to compile the policy. This is done once per policy and platform pair, and not experienced by the end user. While it is important that the time required to compile a policy is not excessive, performance is not a great concern since policy compilation is done infrequently. Next, the application transformer prepares a particular application to enforce a policy. This is done once for each application, policy and platform combination. Users experience this time every time they install a new application to run with a policy. If Naccio were integrated into a web browser, it would be experienced for each new applet or control encountered. Hence, it is important that the application preparation time is low enough that it is not noticeable to the user. Finally, there is the performance overhead when the transformed program is running. This is necessary whenever the program is running with a policy enforced. The overhead should be commensurate with the complexity of the policy. It is unacceptable to have high overhead when enforcing a simple policy, but reasonable for the overhead required to enforce a complex and ubiquitous policy to be high. The rest of this section introduces some policies for testing and discusses the performance properties of Naccio in each of these phases.

8.5.1 Test Policies

For the experiments, the following policies are used:

Null is an empty policy that does no checking. This is a baseline to measure the overhead required for no checking.

NoBashingFiles is defined in Figure 5. It disallows any destructive manipulation of existing files are reports file names in error message.

NoBashingExceptTmp is the property combination from Section 3.2.3. It weakens the NoBashingFiles property to allow modification of existing files in the /tmp/ and /u/evs/tmp/ directories.

LimitWrite is defined in Figure 7. It disallows modifying existing files and places a limit on the number of bytes that may be written.

NetLimit is a policy that uses the NetLimitSendRate property from Figure 14 to limit the network send rate by delaying transmissions. For testing purposes, it sets the limit parameters high enough that it is never exceeded.

SoftSendLimit uses the SoftSendLimit property from Figure 15 to limit the network send rate by splitting up and delaying transmissions using an altered platform interface. For testing purposes, it sets the limit parameters high enough that it is never exceeded.

DisallowAll issues a violation for every resource manipulation.

DisallowAllExcept weakens DisallowAll with permissions that allow common system properties to be observed.

MimicJDK mimics a JDK SecurityManager policy by calling the same check methods as the Java API does. Naccio can be used to mimic any JDK policy using the MimicJDK policy simply by setting the appropriate SecurityManager when the policy is initialized. For our experiments, we use a SecurityManager that performs no checking. Although it reports no violations, it performs differently from the Null policy since Naccio cannot optimize out unnecessary wrappers and resource calls for the MimicJDK policy. Naccio does not analyze the security manager (which can be installed dynamically), so there is no opportunity to optimize out unnecessary checking code.

JavaApplet duplicates the policy HotJava 1.1 enforces on untrusted applets. Rather than using MimicJDK, the JavaApplet policy implements the HotJava policy directly by moving the checking code from AppletSecurity security manager into the safety policy and making the few changes necessary to convert Java code into safety policy actions. This produces a more portable policy, and allows Naccio to eliminate unnecessary work. The JavaApplet policy disallows reading, writing and observing files except as permitted by access lists in the user's configuration file. It only allows network connections to the originating host. Since we run our experiments are applications from the command line, we set the originating host using a command-line definition.

Paranoid is a comprehensive policy that would be suitable for untrusted programs. It includes the NoBashing and LimitBytesWritten properties, as well as properties that limit the number of new files that may be created, limit how many files may be observed, limit the total number of bytes that may be read, restrict the directories that may be read from, prohibit network use, and constrain the creation of windows and manipulation of threads.

TarCustom is a policy designed specifically for the tar archive utility. It instantiates several properties specifically targeted to the tar application, as well as some general properties, such as the NoNetwork property that disallows all network use. It includes a property that allows one file with a name ending in .tar to be overwritten if the c flag is used to create an archive, but allows no other files to be overwritten. TarCustom also limits the number of bytes written at all execution points to a function of the number of bytes read, and restricts files that may be read during the execution to those listed on the command line. In addition to offering protection from malicious or buggy implementations, the TarCustom policy provides protection from user mistakes. For example, executing tar cf * with TarCustom enforced on tar results in a policy violation. With the original application it would replace the first file in the directory with an archive of all other files.

8.5.2 Policy Compilation

The time to compile a policy depends on the size and complexity of the policy, the size of the platform library that must be analyzed and rewritten, and the optimizations done by the policy compiler. This section considers the costs associated with compiling each of the test policies using Naccio/JavaVM. To produce these results, we set options to the policy compiler to turn on all checking optimizations and to produce a policy-enforcing library without renaming classes.

This is the normal case, except in deployments where multiple policies need to be supported simultaneously.

The results for Naccio/Win32 are similar but less relevant. Since Naccio/Win32 does not include a complete Win32 API platform interface, only the policies that deal exclusively with the file system could be compiled correctly. The compilation times for Naccio/Win32 are lower that for Naccio/JavaVM, since it does not need to alter the library classes but only produces the resource implementations and headers and compiles the platform interface file.

Table 1 reports the number of resource operations that need to be implemented to enforce the policy (that is, how many resource operations were determined to do meaningful checking); how many API routines are wrapped; the size of the policy-enforcing library (both the altered API classes and the resource implementations); and the time required to compile the policy. As expected, the Null policy requires no resource operations since there is no checking required. The NoBashing and NoBashingExceptTmp policies both require eleven resource operations – one for the RFile constructor to track file names according to the FileNames state block, and ten corresponding to the members of the RFileSystem.modifyExistingFile resource group. The DisallowAll and DisallowAllExpect policies issue violations for every resource operation in the standard resources. Both policies implement all 122 resource operations provided by the standard resource library. For DisallowAllExcept, a larger policy-enforcing library is produced because of the violation codes needed to track permissions as well as the extra checking code in permission actions.

The number of routines wrapped depends on the implemented resource operations, but one resource operation may require dozens of wrappers if there are many different API routines that manipulate the same resource. The 21 wrappers required for the Null policy comprise the wrappers necessary to guaranteed low-level integrity of the checking. These are the wrappers that protect dynamic class loading and reflection as described in Section 6.2.1. Although these wrappers are not strictly necessary for the Null policy, since it places no constraints on program

Policy	Implemented resource operations	Wrapped routines	Policy-enforcing library size (KB)	Rules in policy description	Compilation time (seconds)
Null	0	21	244	3	126
NoBashing	11	65	263	3	149
NoBashingExceptTmp	11	65	267	4	215
LimitWrite	13	80	280	6	153
NetLimit	10	40	283	4	146
SoftSendLimit	10	40	258	4	146
DisallowAll	122	182	362	26	188
DisallowAllExcept	122	182	373	27	259
MimicJDK	51	139	306	7	225
JavaApplet	43	130	310	6	241
Paranoid	59	140	383	10	234
TarCustom	26	101	316	13	192

Table 1. Policy compilation costs.

Time is the average wall-clock time over three runs. All the results use Sun’s JDK 1.1.7 with no JIT compiler on a 500 MHz Pentium III with 256MB running RedHat Linux 5.2.

behavior, they are required for any policy that imposes behavioral constraints on executions. Naccio does not attempt to optimize out checking necessary for low-level code safety, since it is required for any policy that places any constraints on executions. The other policies require these wrappers, and additional wrappers depending on the resource operations. The DisallowAll policy requires 182 wrappers. This is the highest number of wrappers possible with the standard resources, since all resource operations are meaningful. The only way more wrappers would be needed, is if an extended safety policy altered the platform interface to define additional resource operations.

The size of the policy-enforcing library depends on how much of the API needs to be modified and how many resource operations are required. In the worst case, Naccio would need to copy the entire API. For the normal case, however, only a subset of the API classes need modifications and Naccio need only generate those classes. For all the test policies, the size needed represents less than 4% of the size of the Java API (about 9 megabytes for JDK 1.1.7). If the policy compiler options were set to produce globally renamed library classes to support multiple simultaneous policies as described in Section 5.4.1, the policy compiler would need to rewrite all Java API classes to replace the names.

The policy description file contains the transformation rules that encode what the application transformer must do to enforce the policy. All policies have a rule that gives the location of the policy-enforcing library. Additional rules are needed for each wrapped native method and for each initializer and terminator required. For the Null policy, there are three rules: one gives the location of the policy-enforcing library, and two describe wrapped native methods (the java.lang.Class.forName and java.lang.reflect.Method.invoke methods that must be wrapped to protect integrity of the checking). Other policies have additional rules for wrapped native methods, and calls to initializers and terminators. An additional rule is needed for policies that have permissions (NoBashingExceptTmp and DisallowAllExcept) to indicate to the program transformer that violation codes must be passed to the initializers and terminators.

The final column gives the time needed to compile each policy. The prototype implementation is very inefficient, so it is expected that these times could be significantly improved without substantial effort. The measurements are for Java code running completely interpreted, so a substantial improvement is possible simply by using a native Java compiler. The policy compilation time increases with the number of implemented resource operations and number of routines that are wrapped. The policies that contain permissions (NoBashingExceptTmp and DisallowAllExcept) require violation codes and involve additional processing time.

On average, about half the total time is spent generating wrapper classes and most of the remainder is spent compiling the generated resource implementation. The time spent generating wrappers depends on the number of wrappers required and the performance of the class transformation engine. While the prototype implementation does a reasonably good job of only wrapping routines that need wrappers, the performance of the transformation engine could be significantly improved. The time spent producing the resource implementation source files is minimal, but the time spent running a Java compiler to produce corresponding class files represents about half the policy compilation time. One way to improve this would be to be more selective about which resources are implemented. Naccio/JavaVM generates and compiles a resource implementation even if a resource has no implemented operations. Another option would be to use a faster compiler, or to directly generate class files for resource implementations instead of producing and compiling source files. Since the intermediate representation is available, Naccio/JavaVM should be able to produce class files directly much more quickly than the time required producing source files and running a Java compiler.

Policy compilation is slow, but not a serious concern. It is clear that it could be several times faster in an industrial implementation. Further, policy compilation is a relatively infrequent task. There are ways to avoid the entire compilation process when a policy is being developed. For example, we can generate the unoptimized platform interface library once and only need to produce new resource implementations to compile the policy.

8.5.3 Application Transformation

The time required to transform an application is important, since users experience it every time a new program is run with a safety policy. Table 2 shows results from using Naccio/JavaVM to transform some test applications with the LimitWrite and DisallowAllExcept test policies. The test applications are:

- **jlex** – a lexical analyzer generator available from www.cs.princeton.edu/~appel/modern/java/JLex/.
- **tar** – an implementation of the tar file archiving utility from www.ice.com.
- **ftpmirror**– an application that uses jFtpClient from www.1hostplus.com/java/ to mirror an ftp directory by retrieving a set of files from one site, storing them in local files, and putting them on another site.

Most of the application transformation time is spent reading and writing class files. The application transformer's performance could easily be improved in an industrial implementation. In particular, we can reduce the overhead of application transformation to nearly zero by integrating it into the byte code verifier. The actual work needed to transform an application is limited to some simple string replacements in the constant pool at the beginning of each class file and for some policies inserting a few calls to initializers and terminators into the main method.

The constant pool changes are necessary to handle wrapped native methods. The LimitWrite policy wraps the native java.io.FileOutputStream.write(int) method, so references to this method in the application class files need to be replaced with references to w_write. Since ftpmirror and jlex do not call java.io.FileOutputStream.write(int), no changes to the constant pool are necessary.

The instructions added are only for calling initializers and terminators. Since the LimitWrite policy has no implemented intializer or terminator resource operations, no instructions are added to enforce it. Both tar and ftpmirror have a main method that has one exit point; hence, the DisallowAllExcept needs to insert instructions to call each initializer and terminator once. The jlex application has a return statement in the middle of its main method, so Naccio/JavaVM must insert additional calls to the terminators before this return.

Program	Size of application classes (KB)	LimitWrite			DisallowAllExcept		
		Constant pool changes	Instructions inserted	Time (seconds)	Constant pool changes	Instructions inserted	Time (seconds)
jlex	86.7	0	0	1.62	37	26	1.77
tar	23.4	2	0	1.03	41	21	1.44
ftpmirror	7.1	0	0	0.77	39	21	1.08

Table 2. Program transformer results.

8.5.4 Execution

Assuming the policy generation and application transformation costs are acceptable, the most important cost of enforcing a safety policy is the run-time overhead experienced when the program is run. This section looks at the run-time performance of executions of programs transformed by Naccio/JavaVM to enforce the test policies. To isolate the costs of the safety checking, we first consider some micro-benchmarks that are toy applications designed to do little real work. Then, we report on results for more realistic benchmarks based on the test applications used in Section 8.5.3.

Micro-benchmarks

To obtain an accurate estimate of the overhead required for safety checking, we use two micro-benchmarks:

- **setproperties** runs a loop that calls System.setProperties (null) ten million times. We use setProperties since it is the least expensive operation in the JDK that includes a security check.
- **exists** creates a java.io.File object and runs a loop that calls java.io.File.exists () one million times on that object.

These benchmarks are not intended to correspond to typical programs, but rather to provide a way to isolate the performance overhead associated with safety checking. Hence, they do very little real work relative to the amount of safety checking compared to typical programs.

To test the benchmarks we use policies that do not do any actual checking, but measure the overhead that would be associated with different ways of enforcing a safety policy. These micro-benchmarks are used to measure the overhead associated with introducing checking code, isolated from the cost of actually doing checking. We run each benchmark imposing the following policies:

- **nochecking** – This corresponds to Naccio enforcing a policy that does not constrain the relevant resource operation (either RSystem.setProperties or RFileSystem.observeExists). Of the test policies, Null, NoBashing, NoBashingExceptTmp, LimitWrite, NetLimit, and SoftSendLimit are equivalent to nochecking for the micro-benchmarks, since they place no constraints on either RSystem.setAllProperties or RFileSystem.observeExists.
- **emptycheck** – Naccio enforcing a policy that has resource operations for RSystem.setAllProperties and RFileSystem.observeExists that do no work. Normally, Naccio would optimize out these resource operations and remove the related wrappers; for this benchmark, we configure Naccio to prevent these optimizations so that we can measure the overhead associated with the resource operation call.

For both policies, the policy compiler removes code associated with calling the JDK security manager, as described in Section 5.4.1.

There results are compared to setting the JDK security manager to either null or an empty manager:

- **JDK-null –** Standard Java execution with the SecurityManager set to null. This reflects the normal execution for a Java application.

- **JDK-empty** – Standard Java execution with a SecurityManager that does no checking. This reflects the execution of a Java applet with a SecurityManager installed but a policy that does no relevant checking.

Table 3 shows the time spent in the micro-benchmark loop for each Naccio policy or JDK security manager setting. The results give an indication of the relative costs of different ways of interposing checking code. Both Naccio policies require less overhead than is required for either JDK security manager setting, since they do not need to obtain and test the security manager. The traditional JDK security approach requires obtaining that security manager (either by calling System.getSecurityManager or referencing of a local instance variable in java.lang.System methods), and a comparison to null and a branch. The setproperties micro-benchmark runs 23% slower using the null SecurityManager because of this code. The exists benchmark requires more work to obtains the security manager since it needs to call System.getSecurityManager while setproperties can reference an instance variable. Nevertheless, the relative overhead is less since the java.io.File.exists method does substantially more work than java.lang.System.setProperties.

The results for emptycheck and JDK-empty reveal that the overhead associated with calling security checks is lower with Naccio/JavaVM than using a JDK security manager. This is a result of saving the overhead associated with retrieving and testing the security manager, and that the security manager calls being virtual method invocations and the Naccio resource calls being static method calls.

Since the micro-benchmarks isolate same security-relevant code excerpts, they should not be used as a guide to overall program performance. They do indicate, however, that there is some non-negligible cost associated with JDK-style checking even when the SecurityManager is null. Further, Naccio's approach of inserting checking code when necessary is more efficient than the fixed checks included in the Java API. Although the relative costs will vary according to the virtual machine used, even an ideal compiler would not be able to avoid this overhead using standard JDK security mechanisms since the result of System.getSecurityManager is not guaranteed to be fixed over an execution.

	setproperties		exists	
Policy	Time (s)	Time (ratio to nochecking)	Time (s)	Time (ratio to nochecking)
nochecking	2.67	1.00	5.61	1.00
JDK-null	3.30	1.23	6.06	1.08
emptycheck	2.83	1.06	5.95	1.06
JDK-empty	5.77	2.16	6.44	1.15

Table 3. Micro-benchmark performance.

Time is the average time over ten trials measured on the system clock before and after the microbenchmark loop using Sun's JDK 1.1.7 with no JIT compiler on a 500 MHz Pentium III with 256MB RAM running RedHat Linux 5.2.

Program benchmarks

The costs of enforcing a policy on an execution depend on both how much checking is done and how expensive it is relative to the other work done by the program. Here we look at the relative costs of enforcing the test policies on different program benchmarks using the programs introduced in Section 8.5.3. The benchmarks are:

- **jlex** – running JLex on a 700-line sample file.
- **tar** – running tar to create an archive of a directory tree containing 1736 files and 5.2 megabytes of data.
- **ftpmirror** – running ftpmirror to mirror ten 1-megabyte files from an ftp server on the local network to a different location on the same ftp server.

Table 4 shows the number of calls to resource operations and number of violations reported for each benchmark execution. The number of calls to resource operations gives an indication of how much checking is done for a given execution. The actual work associated with each resource operation call varies depending on the policy, but the number of calls gives a good indication of how comprehensive the checking is.

The Null policy requires no resource calls and issues no violations since it does no checking. The NoBashing and NoBashingExceptTmp policies call resource operations to construct RFile objects for each file used in the execution. For the tar benchmark, there are 1737 file objects corresponding to the 1736 files in the directory tree being archived and the single output file. The additional resource call is the one call to RFileSystem.openOverwrite for the output file (which exists before the execution starts). It does the checking associated with the modifyExistingFile group and issues a violation before the output file is overwritten. The LimitWrite policy requires these calls and additional calls to preWrite and postWrite for the API call that writes to the output file. The DisallowAll and DisallowAllExcept policies associate checking code with every resource operation in the standard resource library. DisallowAll issues a violation for every operation except the initialize and terminate operations for RSystem; as a result a large number of violations are issued for the tar and ftpmirror benchmarks that do a lot of resource manipulations. For the DisallowAllExcept policy, some of these violations are overridden by allow commands.

Policy	jlex		tar		ftpmirror	
	resource calls	violations	resource calls	violations	resource calls	violations
Null	0	0	0	0	0	0
NoBashing	3	1	1738	1	31	10
NoBashingExceptTmp	3	1	1738	1	31	10
LimitWrite	63	1	3826	947	13021	5211
NetLimit	1	0	1	0	1093	0
SoftSendLimit	1	0	1	0	1309	0
DisallowAll	127	125	57099	57097	30819	30817
DisallowAllExcept	127	110	57099	57097	30819	30817
MimicJDK	9	0	13896	0	179	0
JavaApplet	10	2	13896	0	166	0
Paranoid	87	28	31971	30881	6223	12490
TarCustom	88	1	25792	1	6221	12520

Table 4. Benchmark checking.

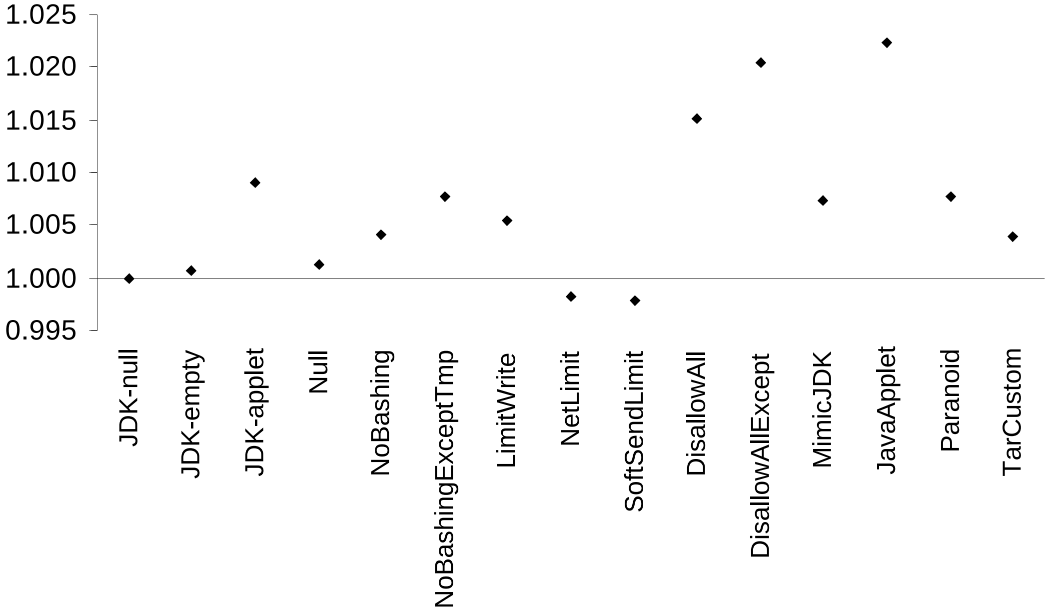


Figure 24. Results for jlex benchmark.

Value is execution time using the policy shown, averaged over 50 trials. Times are divided by average execution time using JDK-null for that benchmark to show the relative overhead.

For the performance measurements, we modify the policy compiler to remove the actual violation production. Otherwise, the overhead is dominated by creating strings for violation messages. Since in normal situations execution would be terminated after the first violation, this is a reasonable thing to do for generating performance measurements for policies that would issue multiple violations. For comparison, we use the JDK-null, JDK-empty and JDK-applet policies. The JDK-null and JDK-empty policies were introduced in the previous section – JDK-null sets the security manager to null, and JDK-empty sets the security manager to a SecurityManager that does no checking. The JDK-applet policy sets the security manager to the AppletSecurity security manager (version 1.76) that is used by HotJava 1.1. We modify AppletSecurity to enforce the same policy on applications as it does on applets (by changing the return value of one function) since all the test benchmarks are applications. To avoid any security violations, we set the acl.read and acl.write properties to allow the necessary reading and writing, and set the originating host to allow the network connections made by the ftpmirror benchmark. The JavaApplet policy enforces the same policy as JDK-applet using Naccio security mechanisms.

Figures 24-26 show the execution results for each benchmark. The checking overhead for jlex and ftpmirror is low compared to that for tar. This results from the difference in the ratio of security-relevant operations to the amount of real work done by the different benchmarks. For each benchmark, the overhead varies for each test policy depending on the amount of checking work done by the policy.

For the jlex benchmark, the security overhead is virtually negligible. At most, it is just over 2% for the JavaApplet policy. The low overhead is not surprising since jlex executes few security-related operations compared to the amount of processing it does. The JavaApplet result compares unfavorably to the JDK-Applet result for the same policy, although the absolute differences are very small. Both policies require the same initialization code that reads a file that contains the access permission settings. This explains the bulk of the overhead. The rest is checking associated with the file opens. JavaApplet has to do the additional work of maintaining abstract resource objects associated with the files, although second order effect like caching may be enough to explain the performance differences.

The tar benchmark requires far more security overhead than jlex. The checking overhead for the tar benchmark ranges up to 250% for the JDK using the JavaApplet security manager; for all other policies the overhead is below 70%. The reason the JDK-applet performs so poorly is that it creates a new java.io.File object and calls getCanonicalPath for each security check call. Since the SecurityManager.checkRead method takes a String parameter, it does not have access to the corresponding java.io.File object even if it has already been created. In the checking code, checkRead needs to convert the String to a canonical path for checking. This is done by calling, new java.io.File (file).getCanonicalPath (). Both the file creation and the getConnonicalPath calls are expensive. For each file that is added to the archive, tar calls java.io.File.isDirectory twice, java.io.File.length, java.io.File.lastModified, and the java.io.FileInputStream constructor that actually opens the file. Each of these calls the SecurityManager.checkRead function and incurs the costs of creating a new file object, calling getCanonicalPath and scanning the access list to determine if reading if permitted. As a result, the benchmark takes 3.5 times as long using the JDK-JavaApplet as without security checking. Using Naccio to enforce the same policy is much less expensive, as is evident from the results for JavaApplet. The overhead is 70%, about a quarter of the overhead required for JDK-JavaApplet. Since the checking code uses an RFile object, the canonical path for the file is stored in an object field using a state block the first time it

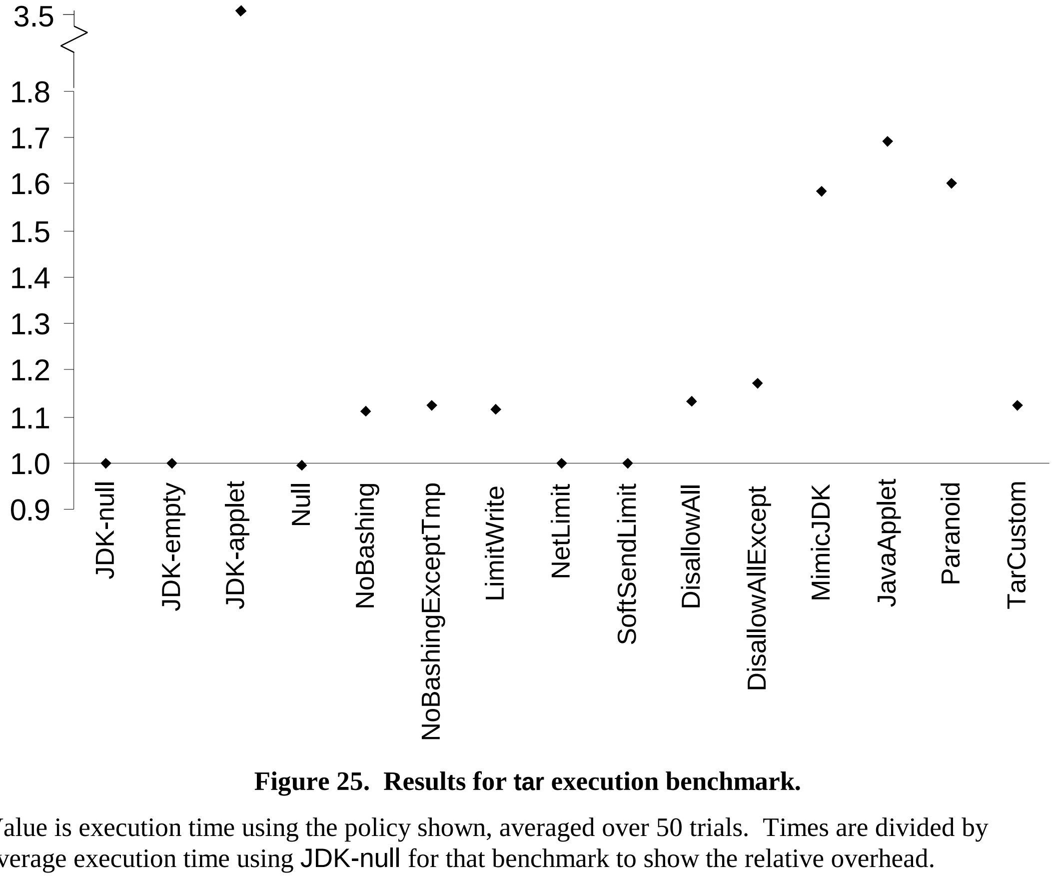


Figure 25. Results for tar execution benchmark.

Value is execution time using the policy shown, averaged over 50 trials. Times are divided by average execution time using JDK-null for that benchmark to show the relative overhead.

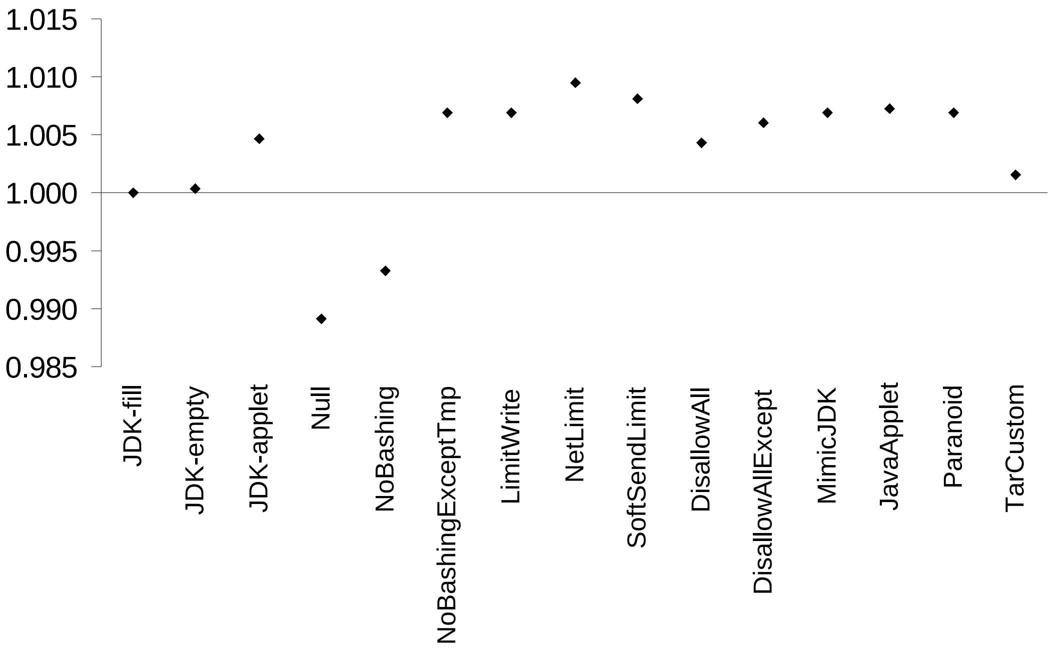


Figure 26. Results for ftpmirror execution benchmark.

Value is execution time using the policy shown, averaged over 50 trials. Times are divided by average execution time using JDK-null for that benchmark to show the relative overhead.

is requested. Instead of doing this operation five times per file archived as is necessary for the JDK-applet, the JavaApplet policy only does it once. It is safe to store this result, since once a java.io.File object is created the pathname it refers to cannot change.

The results for ftpmirror are shown in Figure 26. As with jlex, the overheads are small since the security checking work is small relative to the actual work done by ftpmirror. The execution time is dominated by the time for actually sending or receiving data over the network, so even a complex policy involving substantial checking such as SoftSendLimit can be enforced with less than 1% overhead. For the tests, the limits for SoftSendLimit are set high enough that there is no need to delay network sends, otherwise the execution would slow down noticeably because of delays introduced in sending data over the network.

Summary

Naccio offers two performance advantages over the JDK security approach. Since the Naccio policies are integrated into the application at transform time resource operations are called directly from the wrapped API routines. By contrast, the JDK approach must call java.lang.System.getSecurityManager to obtain a security manager at run time, test if it is null, and make a virtual method call to a security manager check method. The micro benchmarks indicate this overhead can be significant, but it is usually too small to be noticeable in a program that does useful work. The other performance advantage is that whereas the JDK approach must always call a security manager check method regardless of the policy in effect, Naccio only wraps an API routine when that routine manipulates a resource in a way constrained by the policy in effect. This difference is not clearly apparent from the benchmark results because the JDK security manager check methods are so limited. Security checking may only be associated with a small subset of resource manipulations, most of which are expensive enough that the overhead of a security manager check call is not significant. If the JDK supported more extensive check methods, the advantages of eliminating unnecessary checks would be revealed in the benchmark results.

The main performance disadvantage associated with Naccio is the need to maintain abstract resource objects. For example, each `java.io.FileOutputStream` object used in a Java execution that is enforcing a policy that constrains file manipulations maintains an extra field that stores an `RFile` object. In addition to being passed to resource methods, this object has to be constructed and garbage collected. Further, adding an extra field to the structure may result in unpredictable second order effects because of displacement in the cache.

For the most part, the execution performance results are satisfactory. There is some overhead associated with Naccio enforcing a policy, but it is related to the complexity of the policy and comparable to the JDK overhead. Although Naccio does not offer a significant performance advantage over the JDK mechanisms for most policies, it can enforce a large class of policies that cannot be enforced by the JDK mechanisms.

Chapter 9

Future Work

There are several possible directions for future work. This chapter considers work directed at improving the reliability and performance of Naccio implementations; extending the architecture that would allow for a larger class of policies to be defined and enforced; deploying Naccio implementations in real environments; and exploiting Naccio's definition and enforcement mechanisms in areas other than code safety.

9.1 Improving Implementations

While the prototype implementations are useful for conducting experiments and validating Naccio as a proof of concept, neither prototype implementation is good enough to be considered ready for industrial applications. This section discusses some of the work that would be necessary to produce an industrial quality implementation. Section 9.1.1 discusses some things that could be done to provide better assurance that a Naccio implementation is correct. Section 9.1.2 discusses what would be necessary to make Naccio/Win32 into a complete and secure implementation of the Naccio architecture. Section 9.1.3 suggests ways to improve the performance of the policy compiler, program transformer and execution of the transformed program.

9.1.1 Assurance

For a code safety system to be trustworthy, there must be some assurance that it provides the expected security. As discussed in Section 8.1, one of the security vulnerabilities of Naccio is its dependence on a large trusted computing base. An industrial implementation should attempt to reduce the size of the trusted computing base and validate its most critical parts.

The critical part that is most amenable to validation is the platform interface. A malicious program could exploit an error in the platform to manipulate resources without appropriate checking. One approach is to attempt to prove the platform interface is equivalent to some other model of the platform. Verifying the platform interface against the system library requires a model of execution behavior that captures the resource manipulations described by the platform interface. The resource descriptions provide one such model, but they are only useful for comparison if we can describe the system in terms of those resource descriptions. This is in fact what the platform interface does. Obviously, comparing the platform interface to itself is unlikely to produce useful results. Instead, what is needed is a second platform interface that describes the platform at a lower level.

For example, if we had the Naccio/Win32 platform interface that describes the Win32 API calls in terms of the standard resource descriptions, and a second platform interface that describes Windows kernel calls in terms of those same standard resource descriptions, we could attempt to prove for a given Win32 API implementation both platform interfaces will produce the equivalent sequence of resource operations. This could be done using either the source code or object code for the Win32 API. While it is likely to be more difficult to construct a proof from the object code, using the source code means the compiler used to produce the Win32 API must also be trusted. For most Win32 routines, statically determining what kernel calls are made can be done without unreasonable difficulty. Then the sequence of resource calls made by those kernel calls could be derived from the platform interface. The final step is to determine if those calls are equivalent to the calls made by the Win32 API platform interface for the same routine. If the sequence is exactly the same, they are obviously equivalent. It may be possible to argue that sequences that differ are also equivalent, although this will depend on assumptions about the kinds of checking code that may be attached to resource operations. This would provide a useful test of the platform interface and likely uncover some bugs in both the Win32 API platform interface and the Windows kernel platform interface.

A similar approach could be used the test the Java API platform interface. If we are testing a Java implementation for Win32, we could use the Win32 API platform interface as the secondary platform interface and use it to produce the sequence of resource calls made by each native method implementation. This information could be used along with the Java API implementation, to determine the resource operation sequence associated with every API routine. Comparing it to the resource calls made by the Java API platform interface would reveal inconsistencies between the two platform interfaces and the API implementation.

Another approach would be to use an independent formal model that represents a program execution, and map both the platform interface and the API implementation onto that model. This would make most sense if such a model already existed. A suitable model would be a formal specification of a platform API. An advantage of this approach is that the producer of the model does the work of describing the platform API. Having it produced independently also increases the likelihood that whatever errors it has are different from the errors in the platform interface, so inconsistencies are more likely to be detected. Unfortunately, no suitable specification is believed to exist for either the Java API or Win32 API. It if did exist, validation would require producing a mapping from the Naccio resource descriptions to the independent model, and validating their equivalence.

9.1.2 Complete Implementations

While the Naccio/JavaVM prototype is complete enough to be used for security in a hostile environment, the Naccio/Win32 prototype implementation does not completely implement the Naccio design. In particular, is does not include a complete platform interface and does not perform the protective transformations necessary to ensure the checking code is not bypassed or tampered with. Producing a complete platform interface would be a tedious and expensive process. There may be some ways to automate the process using the Win32 source code, however it is unlikely that it could be done without carefully considering every Win32 API function.

Implementing the protective transformations necessary to provide low-level code safety on Win32 is also a major task. Although there are successful SFI implementations that operate on x86 assembly code [Small97, Erlingsson99], there is no known implementation that works on x86 executables. Producing one requires dealing with several additional complications not present when dealing with assembly code including code discovery and handling jumps to the middle of variable length instructions. There is, however, reason for optimism that a suitable SFI implementation will be available in the near future. There is at least one industrial project directed towards this goal [Feldman99]. Further, an industrial implementation of Naccio/Win32 needs to ensure that multiple threads cannot be used to circumvent safety checking. While we believe this can be done using SFI as described in Section 6.2.2, some extensions beyond standard SFI are necessary to provide the needed assurances.

9.1.3 Performance Improvements

The prototype implementations are designed with ease of implementation as a priority. Although the performance results presented in Section 8.4 indicate that even the prototype performance is acceptable in most situations, an industrial implementation could make substantial performance improvements. This section discusses some straightforward ways to improve the performance of the policy compiler, program transformer, and executing application.

Policy compiler

There are several aspects of the policy compiler that could be changed to improve performance. The prototype policy compiler makes several complete passes over the parse tree. These passes could be combined into a single pass to improve performance at the expense of increased complexity. Optimizations are done using inefficient relaxation algorithms that reanalyze the entire policy definition each iteration. These could be substantially improved to be more selective about what must be reanalyzed. The relaxation could keep track of dependency information so that only the relevant parts of the policy definition need to be reanalyzed. Another way to dramatically improve the policy compiler would be to compile the Java code to a native executable instead of running it as interpreted JavaVM code.

Another way to improve the performance of the policy compiler is to provide better options for trading off compilation time and execution performance. The prototype policy compiler focuses on producing a policy-enforcing library with good execution performance, but when a policy is being developed and the policy compiler is run frequently, it is more important to reduce the compilation time. We could do this by reusing an unoptimized version of the wrapped API classes instead of regenerating the platform interface wrappers each time the policy compiler is run. These wrapped classes would assume every resource operation does useful work. We could compile a policy by analyzing only the resource descriptions and resource use policy, and generating resource implementations including empty routines for any resource operation that has no code. This policy would be inefficient to enforce because of the overhead of calling the empty resource operations, but would be quick to produce.

Program transformer

Although the prototype program transformer is fast enough to be acceptable for many environments, an industrial implementation could be significantly faster. Nearly all the cost of the program transformer is spent in reading, parsing and writing the class files. The actual modifications are limited to simple string replacements in the constant pool, except for the application `main` or applet `start` and `stop` methods in the case of initializers and terminators. The prototype implementation uses the JOIE toolkit, which reads and parses the entire class file. For most classes, there is in fact no need to parse the entire class file since all the modifications are in the constant pool. Since the format of the constant pool is well defined and it is always found at the beginning of the class file, a performance critical program transformer could skip reading and parsing the remainder of the class file entirely, and simply copy it as blocks.

The other thing that could be done if performance of the program transformer is extremely critical would be to integrate it into the byte code verifier. Since byte code verification is already required, replacing names in the constant pool during the verification would incur negligible overhead.

Program execution

Performance of the resulting execution is the most important consideration. The prototype implementation is limited to doing simple optimizations that eliminate unnecessary wrappers and resource operations based on a dependency analysis. An industrial implementation could implement more extensive optimizations to substantially improve run-time performance. Section 5.5 describes optimizations that can be done by integrating the resource implementations and platform interface wrappers. These could substantially reduce the performance costs associated with checking. While doing these optimizations automatically would involve some complexity, they could be done without any new compilation techniques. Ambitious optimizations do increases the complexity of the policy compiler, which is part of the trusted computing base. There is a risk that these optimizations would be implemented incorrectly and lead to new vulnerabilities.

9.2 Extensions

Here we consider some extensions to the Naccio policy definition and enforcement mechanisms. Some of the extensions make it easier to define policies that can be defined with the current mechanisms. Other extensions support classes of policies that cannot be defined or enforced with the current design.

Persistence

Naccio does not provide any mechanisms to support policies that depend on more than one execution. It would be useful to define policies that can be applied to multiple executions of the same program or executions of different programs. Naccio provides no mechanisms for persistent policies, although policy authors could write checking code that manually stores and loads persistent data in a secure database. It would be more satisfactory if mechanisms that support persistence were integrated into the Naccio design. One approach to this that could be adopted by Naccio is used by Deeds [Edjlali98].

Deeds uses history-based access control to constrain the behavior of Java executions. Policies are defined using handlers attached to security events. Security events are limited to check methods defined by the `SecurityManager`. An access-control policy is defined by defining a Java class that provides methods corresponding to event handlers and uses instance variables to maintaining an event history. History is persistent across multiple executions of the same program. Persistence is achieved by using a customized class loader that requires that the entire program be loaded statically (it scans for and rejects programs that use dynamic class loading), and creates a secure one-way hash of the program that is stored in the class loader. This history is saved in persistent storage and loaded using the hash value when execution starts. While history-based policies prevent certain attacks that are not detectable on a single execution, it remains to be seen if they are generally useful. Edjlali et al. suggest extending Deeds by using code transformation to support user-defined security events [Edjlali98], and introducing persistence mechanisms in Naccio by using the approach used in Deeds should be fairly straightforward.

Multi-level platform interfaces

It may be useful to combine more than one platform interface to increase the scope or precision of policies that can be defined. It may be useful to use a partial lower-level platform interface to define resource operations that correspond to manipulations that are not visible at the higher-level. It may also be useful to introduce a partial higher-level platform interface to enable policies that refer to higher-level abstractions. In both cases, there are issues to resolve about how checking is done in the presence of multiple platform interfaces. Here, we consider what might be done to use a lower-level platform interface to constrain resources such as memory use and the CPU, and how a higher-level platform interface could be supported to allow policies to depend on abstractions that are not visible to the regular platform interface.

Certain resources are not visible at the level of the platform interface. For example, the Naccio/JavaVM platform interface cannot see memory allocation and CPU usage. To extend Naccio to support policies defined in terms of resources not visible to the platform interface, mechanisms for introducing either a lower-level platform interface or special transformations are necessary. For Java, this could mean supporting a main platform interface at the level of the Java API and a secondary platform interface at the level of individual byte code instructions. A resource corresponding to memory use could be defined in terms of resource operations that are called when an allocation instruction is used. The policy compiler would insert these calls into the policy-enforcing library and generate a rule that instructs the program transformer to insert them into the application code.

Defining a resource corresponding to CPU use it more difficult. One approach would be to call a `processInstruction` resource operation before every instruction. This would allow fine-grain constraints on CPU usage, but would make the modified program run several times slower than the original. Given that the goal of a CPU resource is to support policies that limit CPU consumption, requiring so much additional CPU consumption to enforce it is probably unacceptable. We can provide less fine-grain usage monitoring by batching the checking. It is easy to statically determine the number of instructions that will execute in a basic block (that is, a code fragment that contains no branches except possibly its last instruction). The individual `processInstruction` resource operation calls could be replaced by a single resource operation call at the beginning of each basic block that accounts for all the instructions in the basic block. This supports less precise policies, since CPU consumption for an entire basic block is accounted for before it starts. An alternative would be to modify the execution environment to call a resource operation for every quantity of CPU use by a particular thread. Information about thread resource consumption should be available to the virtual machine, and it could call resource operations every time a relevant threshold is crossed. It would require modifying a virtual machine, but avoids much of the performance overhead and low-level modification necessary to do this checking directly. JRes [Czajkowsik98] illustrates one way of doing this (see Section 7.3.3).

Support for multiple platform interfaces would also be useful in supporting higher-level platform interfaces. For example, suppose we have a platform interface for MFC in addition to the Win32 API platform interface. It would be useful if a policy could be written that would enforce the necessary constraints on all Win32 programs regardless of whether or not they use MFC, but could do more precise checking for programs that use MFC to allow some behavior that would trigger a violation if all the checking were done at the Win32 API level. One way to produce such a system would be to share resources and policies, but add additional resource operations that are called from the MFC platform interface. These resource operations would encode information that is not available to the standard resource operations, such as that a file for opening was selected by the user using a standard dialog box. They give the policy author a chance to express a policy in terms of those resource operations. Another way would be to have separate policies associated with each platform interface, each described in terms of their own (possible different) resource sets. The policies could be combined so that the policy associated with the higher-level platform interface would override the policy associated with the lower-level platform interface. This could be done using a violation code that has a wider scope that the one currently used to support permissions.

Policy interactions

Naccio doesn't support sharing objects amongst code enforcing different policies. In Naccio/JavaVM, different safety policies use different classes for the Java API objects. For example, if a class enforcing one policy passes a `FileOutputStream` object from a wrapped class to a class enforcing a different policy, a type error results. Although the types were identical in the original classes, the program transformer replaced the names of wrapped classes in the transformed applications with different policy prefixes. Hence, there is a type mismatch when the wrapped object is passed. This would be detected as an error by the byte code verifier running on the second transformed class when it is loaded. The situation for Naccio/Win32 is different, but no better. If an application transformed with one policy passes a pointer to a routine in a DLL that was transformed to enforce a different policy, each will use a different version of the system DLLs. The called DLL will use its policy-enforcing system DLLs, but will not have information about the passed pointer that is stored in the application's policy-enforcing system DLLs. For example, consider the situation where an application executable opens a file and passes the associated file descriptor to a DLL that enforces a different policy. When the DLL calls the write routine in its policy-enforcing system DLL, the mapping between the file descriptor and the actual file is not available and neither policy is enforced correctly.

The current Naccio design does not readily support modifications that would support combining code enforcing different policies. There are three sensible options for what it means to pass objects between different security domains. One option would be for the policy enforced on the original application to be enforced on the objects it passes to other security domains. To implement these semantics with Naccio/JavaVM, it would be necessary to create copies of routines that accept objects enforcing different policies that have type names altered to conform to the policy of the caller. This transformation would need to be done when the second class is loaded. The other options are to enforce the intersection of both policies or to only enforce the second policy. Neither of these options can be easily implemented using the current Naccio design. Naccio assumes that policies are known statically when a program is transformed. This model cannot be readily extended to support introducing new policies during an execution.

9.3 Deployment

This thesis does not address issues involved in deploying a Naccio implementation in a real environment. The prototype implementations are run from the command line, and it is up to the user to manually select the policy to enforce. Several issues must be addressed before Naccio can be usefully deployed as part of a web browser or operating system.

Dealing with violations

The prototype implementation deals with violations in one of three ways depending on a command line flag used in program transformation:

1. It pops up a dialog box that reports the violation and offers the user the choice of terminating that execution or continuing normally.
2. It prints a violation message to the standard error stream and terminates execution.
3. It prints a violation message to the standard error stream and continues execution.

The first option is closest to being satisfactory for a typical interactive deployment environment, while the second option is useful for a non-interactive program (such as a server daemon) and the third option is useful for testing policies.

For an industrial deployment, it would be useful to also offer facilities to dynamically alter the policy to avoid future violations. For example, when the first violation of a property that limits the number of bytes written to the file system is reported, it would be useful to allow the user to select to suppress future violations issues by this property, or to change the limit that must be exceeded before the next violation is reported. Otherwise, each write will lead to another violation and require the user to decide to allow the execution to continue. It is possible to write a Naccio policy that only reports the first write violation by keeping a stateblock that tracks how many violations have been reported, however, it would be useful if there were mechanisms that supported this more generally and allowed users to dynamically choose to suppress categories of violations. Adding this support to Naccio is simply a matter of modifying the policyViolation library method. It can maintain a data structure for each property for which a violation is reported, and record user selections on whether or not future violations of the property should be suppressed.

Another useful option would allow the user to choose to continue the execution, but skip the resource manipulation that produced the violation. This existing violation code mechanism could be extended to encode an option to skip the system call. The policy compiler would generate additional code in platform interface wrappers that checks the violation code, and skips the system call if the user selected this option. For system calls that return values, the wrapper would need to generate a suitable replacement value to return. Often, a null object is the best choice; however, it may be useful to extend the platform interface language so alternate return values can be selected.

Dynamically changing policies is more difficult. Since it is not possible to swap the API classes during an execution, support for swapping policies at run-time is complicated and likely to be error-prone. Instead, supporting policies that can be parameterized seems more reasonable. All that is needed is some way to dynamically pass parameter values to the policies. We could do this using a library class that keeps a database of parameters values and provides routines policies can use to access those values. Then a policy author would write a policy to explicitly retrieve parameter values and use them in checking accordingly.

Global resource scope

The current Naccio implementations associate global resources with an entire execution. In the case of Naccio/JavaVM, this means a single global resource applies to all applets running in the virtual machine. As a result, a policy like LimitWrite places a limit on the total number of bytes written by all applets not on the bytes written by a single applet or collection of applets. In a web browser deployment, it may be more appropriate to have separate global resources apply to the applets loaded from different web sites.

We could do this by changing the policy compiler to produce implementations of global resources that are like regular resource objects. Instead of using static methods and class variables, they would use regular methods and instance variables. The generated platform interface wrappers would need to replace calls to global resource methods with calls to a static method that obtains the appropriate resource object for this thread and then invokes a method on this object. The container would need to keep a mapping between threads and global resource objects to return the correct resource object. This would require some additional execution overhead, but would not require substantial changes to a Naccio implementation.

Splitting up global resource accounting, however, would make a deployment susceptible to new kinds of attacks where an attacker has control over applets that the browser assigned to different global resource scopes. There is no easy way to determine which applets can be attributed to the same producer, so assigning different global resources to different applets is risky.

Policy manager

Naccio does not address the issue of deciding what policy should be used on what code. Requiring a user to manually select a policy for each applet encountered or program installed would be too intrusive for most users. Instead, it should be possible to configure a policy manager to automatically select the appropriate policy based on the source of the program. A straightforward policy manager could be created for Naccio similar to the policy manager in Internet Explorer 5.0. It selects a policy based on the source of the program. Programs from remote sites are classified according to their URL.

Policy development environment

The prototypes do not include any tools to help policy authors write, understand and test policies. If policy authoring is meant to be accessible to non-experts, a better environment for developing policies is essential. A policy development tool that is based on selecting parameters from standard policies, but can be extended with user-defined policy definitions, would provide a useful introduction to policy authoring.

Tools to support policy testing are an area for future research. It would be useful to have tools that can automatically analyze policies and answer questions about what one policy allows that a different policy does not, or whether a policy always disallows a certain sequence of system calls. So far, the only way to test policies is to develop test cases that represent things the policy is supposed to either allow or disallow. With suitable test cases, this is likely to detect simple errors in the policy, but it is not sufficient assurance to know the policy means what its author intends.

9.4 Other Applications

Although the focus of this thesis is on code safety, there are a number of other possible applications of Naccio. The described mechanisms provide a way to alter or monitor the behavior of executions that could be useful in addressing many other problems. We discuss a few possibilities here, but this is by no means a comprehensive list.

Debugging

Without any modifications, Naccio can be used to enforce policies that are useful in debugging programs. For example, a policy could be used to confirm the number of bytes sent over the network is a function on the number of bytes read from files, or that every file that is opened is closed before execution terminates, or that all files created in temporary directories are deleted. The policies used for debugging programs can be more precise than the policies enforced on arbitrary programs since the programmer should know a great deal about the expected behavior of the program. In addition, a policy violation is not necessarily a problem but can direct the programmer to examine assumptions about the behavior of the code more carefully.

Naccio becomes more useful for debugging when platform interfaces are written for application-level objects. Then, programmers could express policies in terms of the expected behavior of application routines. They could test return values against expectations that depend on a history of previous calls and other state. This is similar to the common practice of inserting assertions in code, but expressing those assertions as policies and using Naccio to test them has significant advantages. By separating assertions from the code and expressing them at a more abstract level, Naccio makes it possible for the checking policy and code to be written separately, and allows a checking policy to be written at a high level where it can more easily be compared to the program requirements. In addition, it allows debugging information to be portable across platforms.

Auditing

Rather than issue violations, we can write a Naccio policy that records program activity in a log file. The only difference, is instead of violations producing an error message or dialog box, they would record information in a log file. This log can be used for program analysis. If the logging were done at the system level, it would be useful for intrusion detection. The monitoring could also be done in real-time, and interface with a real-time performance monitoring or intrusion detection system. Because of the expressiveness of Naccio's policy definition mechanisms, a policy can limit monitoring to a precise class of events or event sequences.

Behavior modification

Section 4.2.4 introduced a policy that modifies the behavior of a program to delay and split network sends to conform to a specified bandwidth constraint. By altering platform interfaces, it is possible to change program behavior in ways that are not necessarily security related. For example, we could write a policy that saves backup versions of all files before they are overwritten. We could do this by attaching checking code to the `RFileSystem.modifyExistingFile` group that copies the file in question to a backup directory.

Behavior modification leads to a number of legal and ethical issues. While most software licenses strictly prohibit any modification (including in some cases the binary relocation or caching that occurs during normal executions), there are certain kinds of modification that should be permitted and others that should be prevented. Modifications that introduce security checking should be allowed. Modifying a program to alter author and copyright information should be prevented. Preventing certain kinds of program modification could be done using a trusted execution environment that only allows the unaltered, cryptographically signed program to run. When program transformation tools become common, there will be a need for mechanisms to limit the transformations that can be done.

Defeat against Celtic was a crushing blow to Herrera's "invincible" Inter…It was the beginning of the end for catenaccio. Celtic had proved that the Inter defense could be breached. But Herrera refused to accept that tactics were responsible, instead he blamed sweeper Picchi for Inter's crash. Picchi was soon sold to a lower league club Varese, where he claimed: "When things go right it is always Herrera's brilliant planning. When things go wrong, it is always the players who are to blame."

Andy Gray, *Flat Back Four: The Tactical Game*.
Macmillian Publishers Ltd, 1988.

Chapter 10

Summary and Conclusion

This thesis demonstrates that it is possible to define a large class of safety policies in a general and platform-independent way, and to enforce those policies on executions without an unreasonable performance penalty.

10.1 Summary

The contributions of this thesis are in three areas – mechanisms for defining safety policies, an architecture for enforcing those policies, and prototype implementations of that architecture.

Policy definition mechanisms

Naccio defines a safety policy by associating checking code with abstract resource manipulations. The policy definition mechanisms are general enough to describe a large class of safety policies that includes many useful policies. A subset of definable policies is known as standard safety policies. These policies can be defined using a standard resource library, and are portable across Naccio implementations for different platforms. Altering the platform interface allows additional policies to be defined. Extended policies can be used to constrain any manipulation visible at the level of the platform interface.

Naccio's policy definition mechanisms have considerable advantages over other alternatives. By describing policies in terms of abstract resource manipulations, they isolate policy authors from platform details. It is not necessary to know a particular platform API to produce or understand a standard safety policy. Once a standard safety policy has been developed, it can be reused on all platforms for which Naccio implementations are available.

Policy enforcement architecture

The architecture for enforcing policies is based on transforming programs to insert checking code. The enforcement architecture depends on replacing resource-manipulating calls with wrappers that perform checking around those calls. Low-level code safety mechanisms prevent the program code from tampering with or circumventing the checking code.

The enforcement architecture has two advantages over common alternatives. Because it modifies platform library object code directly, it does not depend on availability of source code and is only loosely tied to a particular platform implementation. Second, since it statically analyzes the policy and only introduces wrappers that are necessary for checking, the overhead required to enforce a policy is directly related to the amount of checking it does. If a policy does not constrain a particular resource manipulation, there is no checking overhead associated with that resource manipulation. The main drawback to the enforcement architecture is that it depends on a large trusted computing base. This increases the likelihood that there are vulnerabilities that can be exploited and makes assurance difficult.

Implementations

Naccio implementations have been developed that enforce policies on JavaVM classes and Win32 executables. Naccio/JavaVM is a complete implementation, while Naccio/Win32 does not provide a complete platform interface or implement the protective transformations necessary for low-level code safety. While the prototype implementations are not ready for industrial deployment, they provide a proof-of-concept for the Naccio architecture. The performance results indicate that it is possible to expand the class of policies that can be enforced without sacrificing performance.

10.2 Conclusion

Naccio represents one point in the design space for code safety systems. It is well suited to typical Internet users at small and medium size companies today and for the foreseeable future. It supports enforcement of a large class of policies with low preparation costs and run-time overhead that is minimal for simple policies and scales with the complexity of the policy. By defining policies in terms of abstract resource manipulations, it makes it possible for moderately sophisticated users to define new safety policies. The current design is not well suited to high-security environments because its large trusted computing base makes assurance difficult.

By providing better ways to define safety policies along with efficient and convenient mechanisms for enforcing policies, we hope the situations in which code safety policies are used will be expanded. Currently, code safety is usually considered only for untrusted mobile code. A satisfactory code safety system would be useful in protecting users from bugs in applications from trustworthy sources as well. As the precision of safety policies increases and the costs of enforcement are reduced, policies can be enforced in more situations with more pervasive benefits.

Correction fluid and correcting paper may not be used. If mistakes cannot be corrected through recopying or reprinting the problem page, cross out the mistake and/or insert the new material using a typewriter.

Massachusetts Institute of Technology Specifications for Thesis Perparation 1999-2000.

References

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[Anderson72] Anderson, J.P. *Computer Security Technology Planning Study,* ESD-TR-73-51, Vol. I, AD-758 206. ESC/AFSC, Hanscom AFB, Beford, MA. October 1972.

[Berman95] Andrew Berman, Virgil Bourassa and Erik Selberg. *TRON: Process-Specific File Protection for the UNIX Operating System*. Winter USENIX, 1995.

[Bershad95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, Susan Eggers. *Extensibility, Safety and Performance in the SPIN Operating System*. In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15), 1995.

[CERT96a] CERT Advisory CA-96.20. *Sendmail Vulnerabilities*. http://www.cert.org/advisories/CA-96.20.sendmail_vul.html

[CERT96b] CERT Advisory CA-96.24. *Sendmail Daemon Mode Vulnerability*. http://www.cert.org/advisories/CA-96.24.sendmail.daemon.mode.html.

[CERT96c] CERT Advisory CA-96.25. *Sendmail Group Permissions Vulnerability*. December 10, 1996. http://www.cert.org/advisories/CA-96.25.sendmail_groups.html.

[CERT97] CERT Advisory CA-97.05. *MIME Conversion Buffer Overflow in Sendmail Versions 8.8.3 and 8.8.4*. January 28, 1997. http://www.cert.org/advisories/CA-97.05.sendmail.html.

[CERT99a] CERT Advisories. http://www.cert.org/advisories/.

[CERT99b] CERT Advisory CA-99-02-Trojan-Horses. February 5, 1999. http://www.cert.org/advisories/CA-99-02-Trojan-Horses.html.

[Cnet99a] Data virus forces email shutdowns. Cnet News, June 10, 1999. http://www.news.com/News/Item/0,4,37658,00.html.

[Cnet99b] *Java program crashes Windows 95, 98*. http://www.news.com/News/Item/0,4,0-35760,00.html.

[Cohen98] Geoff Cohen, Jeff Chase, and David Kaminsky. *Automatic Program Transformation with JOIE*. 1998 USENIX Annual Technical Symposium.

[Cohn97] Robert Cohn, David Goodwin, P. Geoffrey Lowney and Norman Rubin. *Spike: An Optimizer for Alpha/NT Executables*. In USENIX Windows NT Workshop, August 1997.

[Compaq99] Compaq Corporation. *Compaq JTrek: Product Information*. http://www.digital.com/java/download/jtrek/index.html. July 1999.

[Cyber97a] CyberMedia. *Internet Privacy and Security: A White Paper*. 1997. http://www.cybermedia.com/products/guarddog/gdwhite.html.

[Cyber97b] CyberMedia. *CyberMedia Announces Immediate Availability of Guard Dog Deluxe*. Press Release, October 8, 1997. http://www.cybermedia.com/company/pr/gddeluxe.html.

[Czajkowsik98] Grzegorz Czajkowsik and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. ACM OOPSLA Conference, Oct 1998.

[Denning80] Denning, P. J. *Working Sets Past and Present*. IEEE Transactions on Software Engineering, SE-6, 1980.

[Detlefs96] David L. Detlefs. *An overview of the Extended Static Checking system*. In Proceedings of The First Workshop on Formal Methods in Software Practice, pages 1-9. ACM (SIGSOFT), January 1996. http://www.research.digital.com/SRC/esc/Esc.html

[Deutsch71] P. Deutsch and C. A. Grant. A Flexible Measurement Tool for Software Systems. In Information Processing 1971: Proceedings of the IFIP Congress). Ljubljana, Yugoslavia, 1971.

[Edjlali98] G. Edjlali, A. Acharya and V. Chaudhary. *History-based Access Control for Mobile Code*. In Proceedings of the 5th Conference on Computer and Communications Security, May 1998.

[Erlingsson99] Úlfar Erlingsson and Fred B. Schneider. *SASI Enforcement of Security Policies: A Retrospective*. In Proceedings of the New Security Paradigms Workshop, 1999.

[Evans96] David Evans. *Static Detection of Dynamic Memory Errors*. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, May 1996.

[Evans99] David Evans and Andrew Twyman. *Flexible Policy-Directed Code Safety*. IEEE Symposium on Security and Privacy, May 1999.

[Feldman99] Mark S. Feldman. *Using Software Fault Isolation to Enforce Non-Bypassability*. Unpublished presentation at IEEE Symposium on Security and Privacy, May 1999.

[Fraser99] Timothy Fraser, Lee Badger and Mark Feldman. *Hardening COTS Software with Generic Software Wrappers.* IEEE Symposium on Security and Privacy, May 1999.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Goldberg96] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer. *A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker*. In Proceedings of the 1996 USENIX Security Symposium, 1996.

[Gong97] Li Gong, Marianne Mueller, Hemma Prafullchandra and Roland Schemers. *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2*. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[Gong98] Li Gong and Roland Schemers. *Implementing protection domains in the Java Development Kit 1.2*. In *The Internet Society Symposium on Network and Distributed System Security*, Internet Society, San Diego, CA.

[Gosling96] James Gosling, Bill Joy, Guy L. Steele, Jr. *The Java Language Specification*. Addison-Wesley Publishing Co., September 1996.

[Hicks97] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. *PLAN: A Programming Language for Active Networks*. November 1997. http://www.cis.upenn.edu/~switchware/PLAN/.

[Keller98] Ralph Keller and Urs Hölzle. *Binary Component Adaptation*. European Conference on Object Oriented Programming (ECOOP '98). Springer Verlang Lecture Notes on Computer Science. July 1998.

[Kozen98] Dexter Kozen. *Efficient Code Certification*. Cornell University Tech. Report 98-1661. January 1998.

[Kramer99] Doug Kramer. Personal email communication.

[LaDue96] Mark LaDue. *Hostile Applets on the Horizon*. http://metro.to/mladue/hostile-applets/HostileArticle.html.

[LaDue99] Mark LaDue. *A Collection of Increasingly Hostile Applets*. http://metro.to/mladue/hostile-applets/.

[Lampson71] Butler Lampson. *Protection*. Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems, March 1971. Reprinted in Operating Systems Review, 8(1): 18-24, January 1974.

[Larus95] James R. Larus and Eric Schnarr. *EEL: Machine-Independent Executable Editing*. Proceedings of the 1995 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), June 1995.

[Lee97] Han Bok Lee and Benjamin G. Zorn. *BIT: A Tool for Instrumenting Java Bytecodes*. USENIX Symposium on Internet Technologies and Systems. December 1997.

[Leveson93] Nancy G. Leveson and Clark S. Turner. *An Investigation of the Therac-25 Accidents*. IEEE Computer, July 1993.

[Liskov81] Barbara Liskov, R. Atkinson, T. Boom, E. Moss, J. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, 1981.

[McAfee99] McAfee Corporation Home Page, http://www.mcafee.com.

[Milner90] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997 (originally published in 1990).

[Mogul87] Jeffrey Mogul, Richard Rashid, and Michael Accetta. *The Packet Filter: An Efficient Mechanism for User-level Network Code*. DEC WRL Research Report 87-2. (Also in Proceedings of the 11th Symposium on Operating Systems Principles, 1987).

[Morrisett98] Greg Morrisett, David Walker, Karl Crary and Neal Glew. *From System F to Typed Assembly Language*. Symposium on Principles of Programming Languages, 1998.

[Nauer63] P. Nauer, editor. *Report on the Algorithmic Language Algol 60*. Communications of the ACM, Volume 6, Number 1, 1963.

[Necula96] George C. Necula and Peter Lee. *Safe kernel extensions without run-time checking*. Second Symposium on Operating Systems Design and Implementation (OSDI), October 1996.

[Necula98] George C. Necula and Peter Lee. *The Design and Implementation of a Certifying Compiler*. Proceedings of the 1998 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), June 1998.

[Nelson91] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall Series in Innovative Technology, ISBN 0-13-590464-1, L.C. QA76.66.S87, 1991.

[NYTimes99a] Virus Disables Hundreds of Thousands of PC's. New York Times, April 28, 1999.

[NYTimes99b] New Fast-Spreading Virus Takes the Internet by Storm. New York Times, March 28, 1999.

[NYTimes99c] New Infection Kills Software Through E-Mail. New York Times, June 11, 1999.

[Pandey98] Raju Pandey and Brant Hashii. *Providing Fine-Grained Access Control For Mobile Programs Through Binary Editing*. UC Davis Technical Report TR98-08. August 1998.

[Pethia99] Richard Pethia. *The Melissa Virus: Innoculating Our Information Technology from Emerging Threats*. Testimony of Richard Pethia, Director, Survivable Systems Initiative and CERT® Coordination Center, before the Subcommittee on Technology, Committee on Science, U.S. House of Representatives. April 15, 1999. http://www.house.gov/science/pethia_041599.htm.

[Pietrek94] Matt Pietrek. Peering Inside PE: A Tour of the Win32 Portable Executable Format, Microsoft Systems Journal, Volume 9, No. 3, March 1994.

[Risks95] RISKS Digest. *Warning on Using Win95* message from Paul Saffo. Volume 17, Number 21. June 1995. http://catless.ncl.ac.uk/Risks/17.21.html

[Romer97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. *Instrumentation and Optimization of Win32/Intel Executables Using Etch*. USENIX NT 97 http://etch.cs.washington.edu/etch/etch-usenixnt/etch-usenixnt.html

[Saltzer75] Jerome H. Saltzer and Michael Schroeder. *The Protection of Information in Computer Systems.* Proceedings of the IEEE, Vol 63, No 9. September 1975. http://www.mediacity.com/~norm/CapTheory/ProtInf/

[Schneider98] Fred B. Schneider. Enforceable Security Policies. Cornell University Technical Report TR98-1664. Jan 1998.

[Small97] Chris Small. *MiSFIT: A Tool for Constructing Safe Extensible C++ Systems.* Third Conference on Object-Oriented Technologies and Systems, 1997.

[Spector99] Larry Spector and Lee Badger. *Porting Wrappers from UNIX to Windows NT: Lessons Learned.* Unpublished presentation at IEEE Symposium on Security and Privacy, May 1999.

[Srivastava92] Amitabh Srivastava and Alan Eustace. *A Practical System for Intermodule Code Optimization at Link-Time*. Digital Western Research Laboratory Technical Report 92/6. December 1992.

[Srivastava94] Amitabh Srivastava and Alan Eustace. *ATOM: A System for Building Customized Program Analysis Tools*. Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation. June 1994.

[Srivastava98] Amitabh Srivastava. Personal email, September 1998.

[Sun96] Sun Microsystems. *The Java Language: An Overview*. http://java.sun.com/docs/overviews/java/java-overview-1.html

[Symantec98] Symantec Corporation, http://www.symantec.com.

[Symantec99] *Understanding Heuristics: Symantec's Bloodhound Technology*. Symantec White Paper Series. Volume XXXIV. http://www.symantec.com/avcenter/reference/heuristc.pdf

[TLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. *Efficient and Language-Independent Mobile Programs*, PLDI '96.

[TracePoint97] TracePoint Technology. *Binary Code Instrumentation for Advanced Software Performance Tools*, 1997. http://www.tracepoint.com/lib/binarycode/white_paper/

[Twyman99] Andrew R. Twyman. Flexible Code Safety for Win32. SM Thesis, MIT. May 1999.

[Wahbe93] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham. Efficient Software-Based Fault Isolation. SOSP '93.

[Wallach97] Dan S. Wallach, Dirk Balfanz, Drew Dean and Edward W. Felten. Extensible Security Architectures for Java. SOSP '97.

[Wallach98] Dan S. Wallach and Edward W. Felten. *Understanding Java Stack Inspection*. Proceedings of the 1998 IEEE Symposium on Security and Privacy, Oakland, California. May 1998.

[Wallach99] Dan S. Wallach. A New Approach to Mobile Code Security. PhD Thesis, Princeton University. January 1999.

[Wichers90] D.R. Wichers, D.M. Cook, R.A. Olsson, J. Crossley, P. Kerchen, K. Levitt, R. Lo. *PACL's: An Access Control List Approach to Anti-viral Security*. Proceedings of the 13th National Computer Security Conference. Washington, DC. October 1990.

[Yellin95] Frank Yellin. *Low-level Security in Java*. WWW4 Conference, December 1995. http://www.javasoft.com/sfaq/verifier.html.