

# Flexible Code Safety for Win32

by

Andrew R. Twyman

Submitted to the Department of Electrical Engineering and Computer Science  
on May 21, 1999, in partial fulfillment of the  
requirements for the degrees of  
Bachelor of Science in Computer Science and Engineering  
and  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

With the growth of the global Internet, users have begun to download and run programs for more different purposes and from more varied sources than ever before. These programs should not be allowed to cause harm to a user's system or data, either as a result of malicious code created by an adversary or buggy code that could cause accidentally. Users may have different ideas of what constitutes harm than the program's authors, so they need a flexible way to specify the capabilities and limitations of untrusted programs.

Naccio is a platform-independent architecture for defining safety policies that describe what a program cannot do. To enforce those policies, programs are transformed to integrate safety checking into their operation at run-time. This thesis presents the design of Naccio/Win32, which applies the Naccio architecture to enforce policies on executables running under Microsoft Windows. A prototype implementation provides a proof of concept, and results presented here provide a demonstration of the effectiveness and efficiency of Naccio/Win32's mechanisms.

Naccio/Win32 provides a greater degree of flexibility than any previous code safety system. Safety policies can be written and enforced with no in-depth knowledge of the system, and are specified as general constraints on program actions, rather than being targeted reactions known attacks. New policies can easily be deployed to adapt to changing security needs or system vulnerabilities. The enforcement of policies through transformation is optimized to minimize the overhead introduced, so that users will not suffer a noticeable loss of performance.

Thesis Supervisor: John Guttag

Title: Professor of Computer Science and Engineering

Co-Supervisor: David Evans

## **Acknowledgments**

I would like to thank David Evans, without whom this project would never have come into existence. Dave conceptualized Naccio, created the Naccio/JavaVM prototype, and has been an invaluable help in all of my work on Naccio/Win32. I couldn't have done this without him.

I would also like to thank John Guttag for his help, advice, and insight along the way. Thanks also to Steve Garland, and the rest of the SDS group for their interest, comments, and suggestions.

Dan Scales of Compaq (formerly DIGITAL) WRL was a great help in obtaining information about ATOM and a copy of it for evaluation. Robert Cohn, David Goodwin, and P. Geoffrey Lowney, also of Compaq, also deserve many thanks for their willingness to help with the use of Spike for Naccio/Win32, as well as their advice on general issues relating to the transformation of programs on Windows NT.

Thanks to Steven Lucco of Microsoft for his thoughts and advice on alternate forms of SFI for use by Naccio/Win32.

Finally, many well-deserved thanks go to my parents and family who got me this far in the first place, and to all the friends who made my time at MIT so enjoyable when I wasn't too busy to enjoy it.

## **For More Information**

The author can be contacted by email at [twyman@sds.lcs.mit.edu](mailto:twyman@sds.lcs.mit.edu). More information on the Naccio project is available at <http://naccio.lcs.mit.edu/>, and questions or comments can be directed via email to [naccio@sds.lcs.mit.edu](mailto:naccio@sds.lcs.mit.edu).

The source-code of the Naccio/Win32 prototype presented here is available to interested researchers, who should send email to [naccio@sds.lcs.mit.edu](mailto:naccio@sds.lcs.mit.edu).

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
<b>2</b>	<b>Naccio</b>	<b>12</b>
2.1	Naccio Architecture Overview . . . . .	12
2.2	Expressing Safety Policies . . . . .	14
2.2.1	Describing Resources . . . . .	15
2.3	Describing Policies . . . . .	17
2.4	Usability and Efficiency . . . . .	19
2.4.1	Usability Concerns . . . . .	19
2.4.2	Efficiency Concerns . . . . .	20
<b>3</b>	<b>Win32 Issues</b>	<b>22</b>
3.1	The Windows Architecture . . . . .	23
3.2	Platform Choice . . . . .	26
3.3	Naccio/Win32 Design Overview . . . . .	28
<b>4</b>	<b>Policy Transformations</b>	<b>31</b>
4.1	Resource Compilation . . . . .	31
4.2	Platform Interface and Platform Compilation . . . . .	34
4.3	Application Transformation . . . . .	36
<b>5</b>	<b>Protective Transformations</b>	<b>38</b>
5.1	Control-Flow Safety . . . . .	40
5.1.1	SFI Background . . . . .	41

5.1.2	Use of SFI in Naccio/Win32 . . . . .	44
5.1.3	Non-Local Control-Flow . . . . .	46
5.2	Memory Safety . . . . .	48
5.2.1	Single-Thread Memory Protection . . . . .	49
5.2.2	Multithreading Issues . . . . .	50
<b>6</b>	<b>Prototype Implementation</b>	<b>55</b>
6.1	The Resource Compiler . . . . .	55
6.2	The Platform Compiler . . . . .	57
6.3	The Platform Interface . . . . .	59
6.4	The Application Transformer . . . . .	60
<b>7</b>	<b>Results and Analysis</b>	<b>62</b>
7.1	Sample Policies . . . . .	62
7.2	Sample Applications . . . . .	64
7.3	Performance Results . . . . .	65
<b>8</b>	<b>Related Work</b>	<b>69</b>
8.1	Low-Level Code Safety . . . . .	70
8.2	High-Level Code Safety . . . . .	72
8.3	Binary Editing . . . . .	75
<b>9</b>	<b>Conclusions</b>	<b>78</b>
9.1	Future Work . . . . .	78
9.2	Summary and Evaluation . . . . .	80
<b>A</b>	<b>Sample Safety Policies</b>	<b>83</b>
A.1	Resource Descriptions . . . . .	83
A.2	Sample Policies . . . . .	85
A.2.1	Safety Properties . . . . .	85
A.2.2	State Blocks . . . . .	88
	<b>Bibliography</b>	<b>90</b>

# Chapter 1

## Introduction

As the usage of computers and the Internet has grown, the amount and importance of data that users keep on their computers has increased, along with the potential risks when that data is manipulated or transmitted by programs. At the same time, the distribution of many and varied programs has become easier and more common. Mobile code downloaded on-demand allows great flexibility in content and capability provided on the World Wide Web. Viewers for new file formats allow users to extend the range of data they can utilize, and plug-ins or applets add new functionality. Users are even beginning to purchase applications online and download them for use.

Users often download code from sources that they do not entirely trust, and should be able to run that code without allowing it to do damage to their system. Even code from reliable sources often contains bugs that could cause damage, particularly if the data being processed by that code could itself have been generated by an adversary. Users may also have a different standard of safe behavior than that of a program's authors. For instance, a program may be written to transmit the user's personal information via the network for registration purposes, while the user may wish to protect his privacy. Code safety is required to prevent malicious or buggy code from causing damage. Researchers have long appreciated its importance [16], and as it has become easier to distribute programs through the global network companies and individuals have become increasingly concerned about what damage a program may cause.

Naccio is a code safety system designed to provide robust code safety while meeting

three key goals: flexibility, usability, and efficiency. Providing code safety in today's computing environment requires greater **flexibility** than modern operating systems provide. There is a need for a code safety system that can protect system resources from malicious or buggy code, without limiting the ability of valid code to do useful work. This requires that different programs be run with varying levels of trust and capability. For instance, a software installer might be limited to using certain portions of the file system and not using the network, while a video-player applet might be allowed network access but no access to the user's files. A primary weakness of code safety systems is that they often fail when attacked in ways their designers did not foresee, so a good system should make it easy for new safety mechanisms to be created and deployed to deal with new threats or security needs. All of this requires that a code safety system be flexible, allowing a wide variety of policies to be defined and enforced.

No code safety system can be effective if it is not used, and thus **usability** is an overriding concern. It must be simple for users to protect themselves, and only require a small amount of effort for administrators to install and configure the system and tailor it to their needs. Code safety must be applied to any programs a user wishes to run, without placing any unacceptable burden or trust on the author of that software to cooperate with the safety system.

In addition, **efficiency** must be a major goal. Users will not use a code safety system if it introduces unacceptable overhead in the time taken to download or run code, and system administrators will not deploy it if it requires unacceptable set-up time, processing, or storage requirements. The safety system must be unobtrusive, and introduce a minimum of delays or additional demands for computing resources. It should certainly have no impact on programs that are not being limited by a safety policy.

Naccio is a flexible platform-independent architecture for defining safety policies and enforcing those policies on programs, designed to meet the three key goals of flexibility, usability, and efficiency. Conceptually, Naccio takes as inputs a program and a description of a safety policy, and produces a new program as output, as shown in Figure 1-1. That trusted program behaves as the original program would until such time as the safety policy is violated, at which point action is taken to respond to the violation.

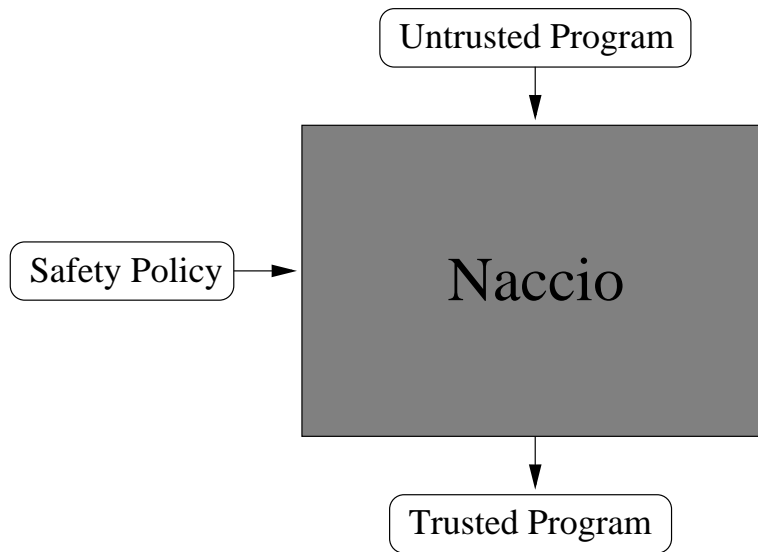


Figure 1-1: Naccio conceptual functionality

Safety policies are defined in a simple, platform-independent way that allows maximal flexibility by allowing new policies to be easily created and used on multiple platforms. Policies are enforced by transforming programs before they are executed. The input program can be in a fully-compiled executable format, eliminating any requirements for the use of safe languages or specialized compilers. There is also no need for disclosure of source code, which allows program writers to protect the intellectual property of their program designs.

Since all safety checking is incorporated directly into the program itself, no special support is required from the operating system or run-time environment. Also, safety checking can be tailored to a specific application and policy, thus eliminating any overhead associated with unnecessary safety checks. Each application can be run under a different policy, making code safety easy to tailor to specific needs. New policies that respond to new threats are easy to deploy by re-transforming affected applications.

This thesis describes Naccio/Win32, a system that enforces Naccio safety policies on Windows programs. A platform for Naccio is defined by the set of system functions controlled by Naccio, and by the binary format of the input program. Implementing Naccio for a specific platform requires a back-end to perform the required transformations in a platform-

specific way, and a platform interface that describes the functions protected. Naccio/Win32 controls Windows programs by placing constraints on their use of Win32 API system calls. The integrity of Naccio/Win32's high-level policy transformations depends on low-level transformations that prevent untrusted programs from bypassing safety mechanisms.

Naccio/Win32 provides a greater degree of flexibility than any previous code safety system. The most commonly used code-safety systems are virus checkers and the Java Virtual Machine. Most virus checkers detect known sequences of malicious code. They are effective at detecting known viruses in programs, but cannot adapt quickly to new threats, and offer no protection from programs that may cause damage for reasons other than being infected. Java provides a virtual machine to run programs in an environment where they are isolated from system resources. While this is often effective at controlling programs, it relies on run-time interpretation or compilation of bytecodes, resulting in extreme performance penalties when compared to native executables. Java's security model also places severe restrictions on the types of actions that can be controlled. Verification of programs can be used to show that they satisfy required properties, as is performed by the Java bytecode verifier [40]. Proof Carrying Code [21] and cryptographic signatures provide alternate verification techniques. Unfortunately, the limitations of automatic proof-generation techniques, and the need for trusted compilers respectively limits their usefulness. Operating system centered approaches can be effective, but system-level safety checks often have a performance penalty on trusted programs as well as untrusted ones. Perhaps the most effective such approach is Generic Software Wrappers [9], which applies checking to API functions as Naccio does but uses a kernel extension rather than transformation to apply the checks. Other transformation-based systems such as SFI [37] [29], Ariel [23], and SASI [36] provide greater degrees of flexibility with less performance overhead than a virtual machine, but do so in platform-specific ways. None of these solutions (analyzed in more detail in Chapter 8) provides the level of flexibility, usability, and efficiency provided by Naccio/Win32.

The rest of this thesis describes the Naccio/Win32 design and presents a prototype implementation that provides a proof of concept. Chapter 2 gives a more detailed overview of the Naccio architecture. Chapter 3 discusses issues that arise in applying Naccio to



the Win32 platform specifically, and introduces the design of Naccio/Win32. Details of that design are presented in two parts: Chapter 4 describes the policy transformations used to enforce policies by controlling system calls and Chapter 5 describes lower-level protective transformations used to ensure that safety checking cannot be bypassed by untrusted programs. Chapter 6 describes a prototype implementation of Naccio/Win32, and Chapter 7 describes results obtained using that prototype to enforce policies on sample applications. Chapter 8 describes previous work in code safety and program transformation as it relates to Naccio. Finally, Chapter 9 concludes and presents ideas for future work.

# Chapter 2

## Naccio<sup>1</sup>

### 2.1 Naccio Architecture Overview

Naccio<sup>2</sup> is a platform-independent architecture for code safety. It allows safety policies to be defined in a platform-independent way and enforces those policies on programs by transforming code.

Naccio transforms programs to produce a trusted program guaranteed to satisfy the specified safety policy. The trusted program is generated by controlling the program's use of system calls. A safety policy is expressed in terms of a platform-independent set of resources. The transformation occurs in two stages, as shown in Figure 2-1. The first stage is performed once for each safety policy, independent of any individual application. The *policy compiler* takes the following inputs:

- **Safety Policy:** A description of the constraints to be placed on the program. Safety policies are expressed in terms of abstract resources, and are platform-independent. For example, a safety policy could specify a limitation on the number of bytes that an application may write to files.

---

<sup>1</sup>Much of the platform-independent architecture of Naccio has been developed separately by David Evans and the author, and is described in [7].

<sup>2</sup>The name Naccio is extracted from *catenaccio*, a soccer tactic where a team defends aggressively near the midfield. Like *catenaccio*, Naccio seeks to protect a system by disarming potential attackers before they are allowed to run rampant on the wrong side of a protection boundary. Naccio is also an acronym for “Never Again Could Code Inflict Outrage” and “No Acronym Can Create Instant Ovations”.

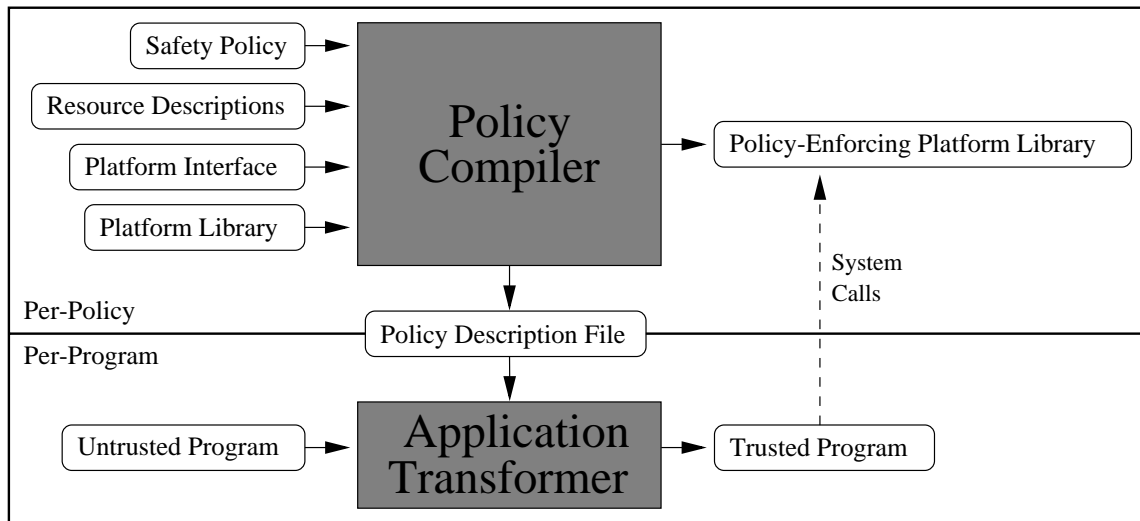


Figure 2-1: Naccio components and operation

- Resource Descriptions:** Abstract operational descriptions of system resources. These are platform-independent, allowing program behavior to be specified and constrained without relying on platform-specific notions. For example, an abstract file system resource would include descriptions of all operations that can be performed on a file system, including a write operation that could be constrained by the policy mentioned above.
- Platform Interface:** An operational description of a particular system platform, such as the Win32 API, that describes the effect that each system call has on abstract resources. For example, the platform interface might specify that a system call to write bytes to a file corresponded to the write operation of the abstract file system, with an argument specifying the number of bytes written.
- Platform Library:** An implementation of the platform library itself. This is the unaltered binary code provided by the platform, such as the Win32 API DLLs or Java API classes.

The job of the policy compiler is to create a policy-specific version of the platform library that performs all safety checking necessary to enforce the policy. Wrapper functions in the

policy-enforcing library perform the necessary safety checks, in addition to performing the work of the original system functions (typically by calling the original functions themselves).

The other product of the policy compiler is a policy description file. This file describes the transformations required to enforce the specified safety policy. Usually this information takes the form of a pointer to the policy-enforcing version of the system library, as well as a description of which system calls must be redirected. The *application transformer* uses this information to transform applications and enforce the policy.

If the trusted program is to be guaranteed to satisfy the specified safety policy it must be unable to affect protected system resources without using the policy-enforcing version of the system library. This requires some lower-level transformations, to ensure that the program cannot bypass the policy-enforcing library. It may also be necessary to reject some classes of programs altogether, either because they can be shown at transform time to violate the safety policy, or because they cannot be transformed safely. For example, self-modifying code is likely to fall in the latter category for most Naccio implementations.

While the Naccio architecture itself is platform-independent, the transformations themselves must be performed in a platform-specific way. The implementation of Naccio for a particular platform includes creating the platform interface, as well as a policy compiler and application transformer specific to a particular platform library, operating system, processor architecture, and executable format. This thesis describes that process for the Microsoft Windows platform, presenting the design and a prototype implementation of Naccio/Win32.

## **2.2 Expressing Safety Policies**

The application transformer produces an alternate version of an application that behaves in the same way as the original application, except that it is guaranteed not to violate the selected safety policy. As well as taking different action in the case of a violation, the transformed application must perform the extra bookkeeping required to implement the checks that detect these violations, and thus the behavior of the trusted program may differ from that of the untrusted program in areas such as running time or memory usage. The behavior constrained by Naccio is defined in terms of a set of abstract resources, which

are platform-independent notions of platform-dependent system resources. The way in which these are defined is described in Section 2.2.1. Examples of resources that can be described include a file system, a network connection, an environment variable, or a thread. Safety policies are specified as limitations on the manipulation of these resources, using the method described in Section 2.3.

## 2.2.1 Describing Resources

Resource descriptions provide a way to identify resources and the ways they are manipulated. They are generally platform-independent, but can be used to describe platform-specific resources such as the Windows registry. Resources are described by listing their operations, which identify the ways a resource can be manipulated. Resource operations have no implementation; they are merely hooks for use in defining safety policies.

The meaning of a resource operation is indicated by its associated documentation. The essential promise is that a transformed program will invoke the resource operation with the correct arguments whenever the documented event occurs. It is the job of the policy compiler and platform interface to ensure that this is the case.

Figure 2-2 shows two sample resource descriptions. The `RFileSystem` resource corresponds to the file system. The `global` modifier indicates that only one `RFileSystem` resource exists per execution. Resources declared without a `global` modifier are associated with a particular run-time object. Most of the `RFileSystem` operations take an `RFile` parameter, a resource object that corresponds to a particular file.

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

```

global resource RFileSystem
  operations
    initialize ()
      "Called when execution starts."
    terminate ()
      "Called just before execution ends."

    openRead (file: RFile)
      "Called before a file is opened for reading."
    openCreate (file: RFile)
      "Called before a new file is created for writing."
    openWrite (file: RFile)
      "Called before an existing file is opened for writing."
    openAppend (file: RFile)
      "Called before an existing file is opened for appending."
    close (file: RFile)
      "Called before a file is closed."

    write (file: RFile, n: int)
      "Called before n bytes are written to a file."
    preRead (file: RFile, n: int)
      "Called before up to n bytes are read from a file."
    postRead (file: RFile, n: int)
      "Called after n bytes were read from a file."

    delete (file: RFile)
      "Called before a file is deleted."
    makeDirectory (file: RFile)
      "Called before creating a directory."
    //... copy and rename operations elided

    observeExists (file: RFile)
      "Called before revealing if a file exists."
    observeLastModifiedTime (file: RFile)
      "Called before revealing when a file was last modified."
    setLastModifiedTime (file: RFile)
      "Called before setting the last modified time on a file."
    //... other similar observe<X> and set<X> operations elided

resource RFile
  operations
    RFile ()
      "Called to construct a new RFile object."
    finalize ()
      "Called when finished with an RFile object."

```

Figure 2-2: Example resource descriptions (see Appendix A for the elided details.)

```
policy LimitWrite
  NoOverwrite
  LimitBytesWritten(1000000)
```

Figure 2-3: The `LimitWrite` Sample Policy

## 2.3 Describing Policies

Safety policies are defined by attaching checking code to resource operations. Code fragments attached to these operations can perform checks of the validity of the operation, and perform state updates, before deciding whether or not to raise a violation. The class of safety policies that can be expressed using Naccio is thus limited only by the resource operations defined in the abstract resources. For example, we cannot express a policy that enforces precise constraints on CPU usage, since no resource operations (and no platform library calls) correspond to direct computation. Further, Naccio cannot express any liveness properties (i.e. properties that constrain what must happen at some time in the future), though it can approximate some by modifying program behavior to perform specified actions (such as deleting temporary files when a program terminates). Once an execution has reached an invalid state, the typical response is to raise a violation, which for interactive applications provides the user with a message and a choice as to whether to terminate the program or allow it to continue in violation of the policy.

A safety policy is described by listing safety properties and their parameters. Figure 2-3 shows the `LimitWrite` safety policy. It instantiates two safety properties – `NoOverwrite` disallows replacing or altering the contents of any existing file, and `LimitBytesWritten(1000000)` places a million-byte limit on the amount of data that may be written to files. Figure 2-4 shows how these properties are defined.

A safety property consists of one or more `precheck` clauses that identify resource operations and provide action code for enforcing the property. For example, the `precheck` clause of the `NoOverwrite` property identifies the two `RFileSystem` operations associated with opening an existing file for writing (`openWrite` and `openAppend`), the operation associated with deleting a file (`delete`), and the operations associated with

```

property NoOverwrite
  precheck RFileSystem.openWrite (file: RFile),
           RFileSystem.openAppend (file: RFile),
           RFileSystem.delete (file: RFile),
           RFileSystem.setLastModifiedTime (file: RFile)
  //... other set<X> operations elided
  violation ("Attempt to overwrite a file.");

stateblock TrackTotalBytesWritten
  addstate RFileSystem.bytes_written : int = 0;
  precode RFileSystem.write (file: RFile, n: int)
    bytes_written += n;

property LimitBytesWritten (limit: int)
  requires TrackTotalBytesWritten;
  precheck RFileSystem.write (file: RFile, n: int)
    if (bytes_written > limit)
      violation ("Attempt to write more than " + limit + " bytes.");

```

Figure 2-4: Properties used by the `LimitWrite` safety policy (see Appendix A for more properties, and adjustments to include elided operations)

modifying existing files in other ways (`setLastModifiedTime` and similar operations). The action code simply invokes the `violation` command, which will produce a dialog box that alerts the user to the safety violation and provides an option to terminate the program.

The `LimitBytesWritten` property illustrates how a more complex safety property is defined. It takes an integer `limit` parameter, and constrains the total number of bytes that may be written by the application to the value of that parameter. When `LimitBytesWritten` is instantiated in a safety policy, `limit` is bound to a value. To implement the write limit, we need to keep track of how many bytes are written. This is done by the `TrackBytesWritten` state block included by the `requires` clause. It adds a state variable to the `RFileSystem` resource, and attaches a `precode` action to the `RFileSystem.write` operation.

The body of the `precode` action will be run before any checking code associated with the resource operation. Hence, when the `LimitBytesWritten` property check action compares the value of `bytes_written` to the `limit`, the value of `bytes_written` has already been incremented before the comparison. State maintenance and property checking



code are kept separate, since many safety properties may utilize the same state.

## **2.4 Usability and Efficiency**

Naccio and Naccio/Win32 are designed for optimizing usability and efficiency in the most commonly expected scenario, where end users download and run programs using a safety system configured by more knowledgeable system administrators. To further these goals, the system is separated into distinct parts, allowing a separation of effort both on the part of users and administrators, and in terms of the actual computation that must be performed to apply and enforce safety policies.

### **2.4.1 Usability Concerns**

The componentization of the Naccio system allows users and administrators to benefit from the system with a minimum of knowledge and training. Increased familiarity with the Naccio system then leads to greater degrees of flexibility in the safety policies that can be created and enforced. If a Naccio implementation were integrated into a web browser or OS environment, most users would never be aware of its existence unless a default policy is violated. Appropriate transformations can be applied automatically or by administrators when programs are installed. A more flexible system front-end can allow users to choose from several standard safety policies when running programs or downloading them over a network. This will be sufficient in most cases if standard safety policies are sufficiently diverse. For further flexibility, power users and system administrators can define new policies by combining standard safety properties and setting their parameters (as described in Section 2.3), still requiring no detailed understanding of the underlying Naccio policy definition mechanisms.

For interested administrators, the willingness to understand the platform-independent framework of resources and safety policies provides the ability to write specialized safety policies tailored to a specific application or environment. Such safety policies could perform highly specific checking not foreseen by the authors of standard safety policies, such as guarding against accidental misuse of program arguments or adding user-specific

policies or quotas. No administrators need delve into the platform-specific details contained in the platform interface due to the abstraction of the resource descriptions. A single platform interface should be sufficient for each platform. Particularly ambitious system administrators can make relatively minor modifications to the platform interface to provide additional flexibility, through adding support for supplemental APIs, protecting additional system resources, or reacting to violations in ways other than simply alerting users. It is most likely that several of these modified platform interfaces would be distributed with Naccio in a production system.

All of the details required to take advantage of this flexibility are encapsulated in the platform interface and resource descriptions. Thus, it is easy for some users or applications to use functionality provided by alternate resources or platform interfaces without affecting others. The separation also eases the deployment of new policies or patches that address security flaws. All that is required is to replace a single file in a single component of the system, re-compile policies, and re-transform any affected applications, rather than any modifications to the framework and core of the Naccio system.

## **2.4.2 Efficiency Concerns**

It is important to optimize the efficiency of common or repeated tasks, even at the cost of less frequent tasks. Run-time performance is the most important to minimize, since application code can be run many times, and run-time is most noticeable to the user. Transformation time is also important, particularly if the user wishes to download a program, transform it, and run it immediately. Naccio and Naccio/Win32 attempt perform work early, so that it need be performed only once, and at a time that has the least impact on the user. This is accomplished by the separation of responsibility between the policy compiler and the application transformer. The policy compiler performs its work only once for each policy that is to be enforced on a given platform, thus limiting the amount of time spent on the time-consuming compilation to create native code from the high-level description languages of policies, resources, and the platform interface. Because it is run rarely, the policy compiler can spend significant time on optimizations to improve the run-time

performance of safety checks. By performing an aggressive dependency analysis, the policy compiler can eliminate safety checking, resource operations, and platform interface wrappers not required to enforce the selected policy. It can also pass along a list of only the minimal set of required transformations to the application transformer. As a result, system calls that require no checking at all remain untransformed, and those requiring checking introduce the bare minimum of run-time overhead.

The only work that must be done for each application transformed is the redirection of system calls to the versions provided in the policy-enforcing system library, as well as the low-level transformations required to ensure that the untrusted code does not bypass that library. As much work is done at transform-time as possible, saving work at run-time that could potentially be repeated.

The counterpart to this “eager” strategy (performing computation as early as possible to avoid repeated effort) is a “lazy” strategy that delays effort until it is guaranteed to be needed, while still not repeating any effort. Untrusted code used by an application at run-time can be detected by standard wrappers on the system calls used to load code, and the transformer can be automatically invoked on the code as it is loaded. This mechanism would be ideally suited for a program that is downloaded and run only once such as a browser-based applet, particularly if portions of its code are downloaded on-demand.

# Chapter 3

## Win32 Issues

Naccio/Win32 enforces policies on fully compiled Windows executables and their associated dynamic link libraries (DLLs). The platform library protected by Naccio is the Win32 API, as implemented by the system DLLs included with the various Win32 operating systems (such as Windows 95/98, Windows NT, Windows 2000, and Windows CE). Policy-enforcing versions of these DLLs are generated for each safety policy, and application executables and DLLs are transformed to use these alternate libraries.

The transformations performed by Naccio/Win32 can be divided into two categories: *policy transformations* and *protective transformations*. The policy transformations introduce wrappers for standard Win32 API calls to produce a policy-enforcing version of the Win32 API in the form of replacements for system DLLs. Untrusted programs are then transformed to use these DLLs instead of the standard system DLLs. The protective transformations provide the low-level code safety necessary to ensure that programs cannot circumvent the checking in the policy-enforcing library.. The mechanisms used to provide policy transformations are given in Chapter 4, while the protective transformations are described in Chapter 5.

Understanding the platform-specific details of Naccio/Win32 requires some knowledge of the workings of the Windows operating system itself. The following section describes aspects of the Windows architecture that are important to Naccio/Win32. This discussion is primarily based on the Windows NT operating system, which is also the basis of the upcoming Windows 2000. Most details of other Win32 operating systems (Windows 95, 98,

and CE) are similar, and due to the standards of the Win32 API, the policy transformations are applicable equally well to any Win32 operating system, though the small API differences between operating systems must be reflected in the platform interface. There are differences between operating systems (particularly in the memory model) that are relevant only to the protective transformations. Since the current Naccio/Win32 design and implementation is targeted at NT, those differences will not be discussed in detail here.

After this background information has been presented, Section 3.2 discusses trade-offs in the choice of the level at which Naccio is applied to the Win32 platform. Section 3.3 then describes the high-level design of Naccio/Win32.

### **3.1 The Windows Architecture<sup>1</sup>**

Like most modern operating systems, Windows NT has at its core an operating system kernel that interacts with hardware, controls basic operating system functions, and provides a basic set of system calls for use by applications. The primary distinction between kernel and user code (which includes applications as well as many portions of the operating system) is based on virtual memory. Every process run on a system has its own virtual address space, giving each process its own view of memory, which is shared among the potentially many threads of execution within the process. Kernel code has a single view of memory that includes the memory used by all other processes, as well as memory used only by the kernel. An application's access to memory may be limited by page-level protections, but this is usually the case only for portions of memory shared with other applications. User code makes system calls to the kernel by executing a "trap" instruction to invoke kernel code on behalf of the application process.

While the system calls provided by the NT kernel provide full functionality, applications are not intended to use them directly. Instead, applications are written to use one of the OS environments provided by NT: Win32, Win16, OS/2, Posix, or MS-DOS. Each OS environment is provided by a protected subsystem: a user level process that receives

---

<sup>1</sup>Much detail here is based on [5], [24], [25], [26], and [19], which include much more complete details on the NT architecture and executable format.

requests from client processes in the form of messages via the Local Procedure Call (LPC) mechanism, an optimized method of interprocess communication. The protected subsystem receives requests appropriate to the environment it supports, and converts them to NT kernel calls, as well as maintaining data structures specific to the OS environment. This conversion is indirect, as all non-Win32 subsystems actually make calls to the Win32 subsystem, which is the only protected subsystem (in most cases) that makes direct kernel calls.

Since programming in terms of LPC messages would be tedious, Win32 programs make use of a standard application programming interface (API) provided as a set of functions. These are not compiled into application code, but are instead provided in a set of dynamic link libraries (DLLs). As described in the next section, this encapsulation of API code aids in the application of Naccio's enforcement mechanisms.

Both DLLs and executables are stored in the same Windows PE (Portable Executable) file format. This format is used on all Win32 platforms, which makes many of the binary editing tasks performed by Naccio portable. The PE format contains information on the structure of the executable image after it is loaded into memory, as well as supplemental data for use by the loader.

DLLs are not intended to be run as independent programs, but rather to be loaded by applications that need the functions included in the DLL. These functions and their locations within the DLL are included in the DLL's export table. When DLLs are loaded they become available at in the application's memory space. Since much of a DLL is read-only code or data, it can often be shared among multiple applications without the need for more physical memory. The use of DLLs is not limited to API functions. Applications may use DLLs to encapsulate functions, and many supplemental APIs, operating system extensions, and other libraries are provided as DLLs.

When loading an executable image, the loader places it in memory and runs any initialization code included in the image. This provides an excellent opportunity for Naccio to run set-up code for safety checking. The loader also determines addresses as necessary for any external functions that have been implicitly linked. Implicit linking is one of the two methods of utilizing DLL functions. If it uses this method, the executable includes an import table. That table includes a list of the DLLs imported (by name), along with

a list of the functions imported from each DLL (by name or numerical reference). Each named DLL is automatically loaded with the original image. The loader finds the DLL by searching a standard set of paths, including the application's directory, the Windows system directory, and the user's path variable.

As well as simply loading the required DLLs, the loader adjusts references to them so that their functions can be used by the application. To accomplish this all implicitly linked DLL accesses are in the form of indirect jump instructions, which use a target address loaded from the import address table (IAT) in the executable image. The indirect jumps for DLL calls are often compiled directly into the code, but procedure stubs are also included. Those stubs consist entirely of the indirect jump instruction, and allow user code to call the stub address and reach a DLL procedure. Those stubs also provide the address that will be used if user code ever tries to directly access or manipulate the address of an implicitly-linked DLL procedure.

The addresses in the IAT are filled in by the loader. Alternatively, the IAT can be filled in beforehand by the `BIND` utility or the `BindImage` API function, which assume that the DLL loads at its preferred address. If it does, then the loader need not do the work of determining procedure addresses at load time. If it does not, or if a different version of the DLL (with different procedure addresses) is loaded, then the loader recalculates the IAT addresses appropriately.

Rather than using implicit linking, an application may explicitly link DLLs, in which case they are loaded on demand rather than simultaneously with the application executable. To load an explicitly linked DLL, an application calls the `LoadLibrary` API function, which invokes the loader to bring the DLL into the application's address space (searching the same standard path). It stays there until the application uses `FreeLibrary` to unload it. Since there is no IAT for an explicitly linked DLL, the application can call functions in that DLL only through the `GetProcAddress` function, which provides a pointer to the specified function in memory. In Naccio, wrapped versions of these API functions transparently substitute policy-enforcing and transformed DLLs.

## 3.2 Platform Choice

There are two aspects to the choice of the platform described by a platform interface. The first the basic platform determined by the operating system and hardware. Equally important, though, is the position within the hierarchical structure of an operating system where Naccio's transformations are applied. The Naccio platform is defined by the set of system calls protected and libraries trusted, as well as by the actual method of inserting security checks. This positioning is very important for several reasons. First, all code below the level of this platform (such as the system libraries) will be treated as trusted code, and assumed to have the behavior specified by the platform interface. It is only the interface to the platform library (and its behavior) that is important, so the same platform interface can easily be used to protect different versions of the same system library. Naccio/Win32's primary target platform is Windows NT, but the platform can be chosen so as to make the differences between the various Win32 operating systems (NT/2000, 95/98, CE) largely irrelevant.

There are several possible positions in the structure of Windows NT to consider for the Naccio platform interface. The lowest level would be actual instructions and I/O routines of a particular processor architecture. This is not a desirable solution, as enforcing policies at such a low level would require much architecture specific code and modifications to the NT kernel. Any kernel modification would have effects on all programs rather than being limited to programs running with policies.

The next possible level is the NT kernel calls. This is a well-defined API, fitting the desired model of a platform library to be protected by Naccio. Protecting the NT kernel would also mean that the resulting Naccio transformer could be used for all of the OS environments provided by Windows NT. Unfortunately, such an approach would require modifications either to the kernel or to the protected subsystems. Though they run as user code, the protected subsystems are part of the operating system and used by all programs, so again modifications to them would affect all programs, and be specific to NT. They could not be used on Windows 95/98, which do not include protected subsystems. The NT kernel calls are also undocumented, and thus would be difficult to describe accurately in a platform



interface.

The next logical possibility is at the level of Win32 API calls. Within the operating system, the Win32 API is the highest level of interface that is standardized, well-defined, documented [19], and used by a majority of applications. Higher level interfaces (such as the MFC classes or COM objects) could be chosen, but since they are implemented using the Win32 API, there is no compelling reason to locate the platform interface at such a high level and eliminate support for many applications. Choosing the Win32 API limits the Naccio/Win32 transformer to Win32 programs, allowing no support for Win16, MS-DOS, Posix, etc. Since Win32 is the dominant platform for programs running under Windows NT, this limitation seems acceptable.

The Win32 API has the advantage of being entirely encapsulated in DLLs. With the platform interface placed at the level of the Win32 API, system DLLs can be considered trusted code, with wrappers being applied to their entry points by simple modifications to import tables. An additional security benefit of the Win32 API is that most critical system data is stored in the protected subsystems, and is thus inaccessible to untrusted code. Some data is stored locally, and may (for certain uses and policies) need to be protected as part of the protective transformations described in Chapter 5.

Perhaps the primary advantage, though, of identifying the Win32 API as the Naccio platform library is that the wrappers and transformation rules that make up the transformer can be made to be independent of a specific hardware architecture or operating system. As long as the transformation rules apply only to Win32 API calls, then they can be applied to any platform on which the Win32 API is supported. Since Windows NT runs on a number of different processor architectures, this is a significant advantage. Support for other Win32 operating systems such as Windows 95/98 and Windows CE is also possible, though some modifications to the platform interface are necessary due to small differences in API behavior. Executables produced by any compiler can also be supported, since the same interface requirements that allow all code to inter-operate with the Win32 DLLs allow it to be transformed by Naccio.

Only the policy transformations can make full use of this processor and operating system independence. At least some of the protective transformations must be performed at the

lowest possible level to keep untrusted code from bypassing security mechanisms. These transformations can, however, be clearly separated from the actual API wrappers. The wrapper-based policy transformations can be developed and tested independently, and re-used on many different physical architectures. The lower-level protective transformations can then be inserted as a module as appropriate.

### 3.3 Naccio/Win32 Design Overview

Naccio/Win32 separates the policy compiler and application transformer, as prescribed in the Naccio architecture. As shown in Figure 3-1, the policy compiler itself is divided further into platform-independent and platform-specific portions. The platform-independent portion of the system is shared with Naccio systems on other platforms [7] (and written in Java to allow operation on multiple platforms). It consists of the parsers and analyzers for the platform-independent languages for describing program behavior and constraints on that behavior introduced in Section 2.2. The same parsing, analysis, and optimization of safety checking is performed for all Naccio platforms.

In Naccio/Win32 these platform-independent tasks are encapsulated in the *resource compiler*, which reads and analyzes policy and resource descriptions and produces implementations of the described safety checking. The only platform-specific component of the resource compiler is the generation of the code to implement the fully optimized resource operations. This code will be combined with platform interface code and compiled to produce native code that performs the specified safety checks. This is accomplished by a simple translation of the platform-independent language of checking code into C code that can be easily compiled into Windows DLLs. It is the responsibility of the platform interface to ensure that each time an untrusted program performs an operation that could affect resources, the policy-enforcing library makes a call to an appropriate function in these DLLs. The translation of code and construction of these DLLs is described in more detail in Section 4.1.

The resource compiler also produces a description of the results of its analysis of the safety policy. These results are used by the *platform compiler* to determine which wrappers

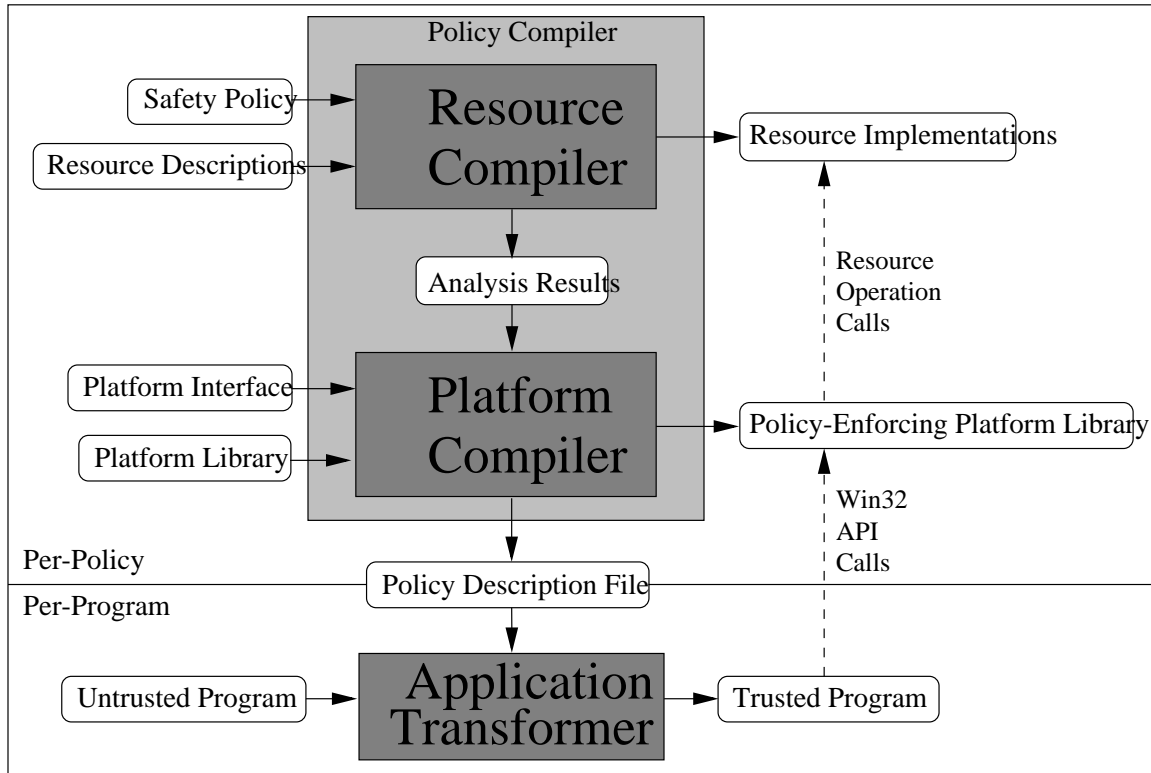


Figure 3-1: Naccio/Win32 components and operation

are necessary. Basing its work on the platform interface, the platform compiler produces wrappers: replacement versions of Win32 API functions that perform safety checking by calling resource operations before or after calling actual API functions.

Wrappers are encapsulated into DLLs with appropriate export tables that allow them to replace the Win32 system DLLs. If these replacement DLLs are located on the standard loading path, the application transformer need only change the DLL names in the import tables of application executables and DLLs in order to replace API functions with wrappers. Since the DLL import/export interface is standardized on all Win32 platforms, this transformation can be performed without any regard to the underlying operating system or processor architecture.

Unfortunately, re-direction of calls is not sufficient to ensure that the policy enforced by the modified system DLLs cannot be violated. Naccio/Win32 must also ensure that a program cannot bypass or tamper with the safety checking mechanisms by violating the abstraction of the DLL interface. This is done by the protective transformations described

in Chapter 5. These include transformations to prevent the program from making direct calls to the original system DLLs and from loading or construction of arbitrary pieces of untransformed code. Additionally, the protective transformations must provide protection against direct modification of data used by safety policy code, or by the API functions. They must also prevent uncontrollable program behavior, such as self-modifying code or direct calls to the kernel. Some protective transformations can be implemented through the use of the wrappers. For instance, memory protection can be enforced by controlling access to Win32 memory protection routines. Some protective transformations, however, must be implemented at the very lowest level – raw machine code – to prevent malicious attackers from exploiting low-level tricks to avoid safety checks. As a result, the design and implementation of the protective transformations is specific to a single processor architecture (e.g. x86 or Alpha).

# Chapter 4

## Policy Transformations

The policy transformations insert wrappers in place of system calls, so that safety checks will be performed as the modified application runs. That task is divided between the three major system components. The resource compiler provides an implementation of the safety checks associated with abstract resource operations. The platform compiler uses the platform interface to produce replacement versions of the Win32 system DLLs that incorporate that safety checking. The application transformer modifies applications so they use those replacement versions, and performs the protective transformations described in Chapter 5. This chapter describes each stage of the policy transformations of Naccio/Win32.

### 4.1 Resource Compilation

The code that implements resource operations and their associated safety checking is automatically generated from the resource and policy descriptions introduced in Section 2.2. The same platform-independent parsing and analyses as in any Naccio implementation are performed, followed by Win32-specific code generation to produce the resource implementations and analysis results needed by the platform compiler.

To maximize the generality and portability of the Win32 code-generator, the code produced is standard ANSI C. Resource types and operations can be handled by standard syntax through naming and argument conventions. These conventions must be followed by the generated resource code and the platform interface. Many of these conventions can

```

typedef struct _RFile {
    String name;
} *RFile;

typedef struct _RFileSystem {
    long bytes_written;
} *RFileSystem;

extern RFileSystem RFileSystem__state;

void RFileSystem__op_write(RFileSystem this, RFile file, int nbytes);
#define RFileSystem_write(file,nbytes) \
    RFileSystem__op_write(RFileSystem__state,file,nbytes)

```

Figure 4-1: Excerpts of auto-generated resource header code

be hidden by `#define` and `typedef` statements located in an automatically generated header file that provides a type name for the resource and function prototypes for the various resource functions. Each resource has a corresponding `struct` declaration that contains the resource state introduced by the safety policy. Each resource operation corresponds to a DLL function that contains C implementations of the policy checking code. Function names encode the resource type as a prefix, and take the resource object as their first argument. Global resources are implemented using a single global variable that holds the state of the global resource and is passed as first argument to resource operations. This maximizes commonality with code for non-global resources and thus eases translation. The generated resource code defines macros as shown in Figure 4-1, allowing platform interface code as well as resource code to call global resource operations without knowledge of this global state variable.

Resource operations declared to be performed once at the initiation or termination of the application are placed into the DLL functions that the Windows loader already runs automatically at such times, so that the appropriate checking code is guaranteed to be run appropriately.

Generating the body of the functions that implement resource operations is a straightforward process of combining and translating code fragments. Some of the more complicated pieces of functionality, such as raising safety violations or requesting external information

(such as the size of a file), are provided by the Naccio library. It includes standard data types and functions for use by policies, implemented separately on each platform. For Windows, the Naccio library is provided as a header file and a DLL that provides the needed functions. It provides data type definitions like the `String` type used by the `RFile` declaration in Figure 4-1, allowing the code-generator to be independent of the specific implementation of those data types.

Some code that is not directly based on the resource and policy descriptions must be generated in resource DLLs in order to handle memory allocation, synchronization, or functionality needed by the protective transformations. Such code can be included in new virtual resource operations introduced by the resource compiler and used by the platform interface. Memory allocation, for instance, is handled by explicit constructor and destructor functions for each resource, which must be called at appropriate times by the platform interface. The code for these operations will be automatically generated and combined with any constructor or finalizer code included in the resource description. Using these operations rather than explicit allocate and free procedures in the platform interface allows the resources to perform reference counting or automatic memory management if desirable. In addition it allows more control over where and how resource objects are allocated, which is useful for some of the protective transformations described in Section 5.2.

Synchronization is another important issue in multithreaded environments such as Win32. It is important that access to resource data be synchronized so that safety checking cannot be circumvented due to race conditions when multiple threads attempt to access the same resource simultaneously. Since knowing where to insert synchronization is a difficult problem to solve automatically, the responsibility of synchronization lies with the platform interface author. The resource implementation provides support for this in the form of acquire and release operations automatically generated for each resource. These operations access a synchronization object included in each resource object (as well as each global resource) to lock the object or resource for exclusive use by the calling thread. The wrappers must lock a resource object before calling any operation that could generate race conditions, and unlock the object after the operation (or set of operations) is complete.

Great care should be taken to avoid deadlock conditions when multiple locks are acquired, and to avoid potential vulnerabilities introduced by insufficient synchronization.

The current prototype implementation of Naccio/Win32 does not include synchronization. Hence, it is vulnerable to race-condition attacks in multithreaded environments.

## 4.2 Platform Interface and Platform Compilation

The platform compiler produces the policy-enforcing library used by transformed applications. This library consists of a replacement for each of the system DLLs that provide the functions of the Win32 API. The platform interface describes the behavior of the Win32 API functions in terms of abstract resources. The platform compiler uses that description to generate wrappers that apply safety checking.

The format of the platform interface files is designed to make the generation of platform DLLs as simple as possible while allowing platform interface authors a sufficient degree of flexibility and clarity. Since the platform interface is written only once by an expert, it is acceptable for it to be somewhat difficult to read and write. The simplest way to do this is to directly provide the source code for the wrapper functions, either in C or in a stylized language translated to C by the platform compiler. Annotations, either in that source code or in a different location, can provide information that would be difficult to provide in source code, such as the name of the DLL being described, lists of functions to be wrapped, and any hints the optimizer may require to eliminate unnecessary code in wrappers.

Compiling wrapper source code into DLLs suitable for replacing the Win32 system DLLs requires two other pieces of special support: name management and support for standard wrappers. Name management is required to allow the wrapper DLL to appear identical to the system DLL that it replaces. It must export all of the functions in the system DLL using the same names. The wrapper functions, however, may need to call the actual API functions, so they cannot have the same names in the source code as the functions they replace. The solution is for wrapper functions to be named internally by adding a standard prefix (`wrapper_`) to the name of the function they replace. As the compiled code is linked into a final DLL, the linker is instructed to construct the export table to include function



names without the prefix.

Linker directives also enable a performance-saving option in the case of null wrappers. Null wrappers occur whenever an API function does not manipulate any resources regulated by the safety policy. In such cases, it would be desirable to simply allow the untrusted program to call this API function directly. This can be accomplished using forwarding. Forwarding allows the function pointer normally stored in a DLL's export table to be replaced by a string containing the name of an alternate DLL and a function within that alternate DLL. At load time, references to the forwarded function are fixed to use the DLL and function referenced by that string. Thus with only a very small additional cost at load-time, all potential run-time cost associated with null wrappers is eliminated.

Since the number of exported functions in the system DLLs is large, and many may not need to be handled as part of a platform interface (if they do not affect protected resources) the platform interface author specifies which functions must receive special handling (such as wrapping) while remaining functions receive some default handling specified by the platform interface. Usually the default is either that they are allowed unmodified as described above, or disallowed entirely by forwarding to a violation function.

The second piece of special support for the creation of replacement DLLs is the inclusion of standardized wrapper code that is not part of the platform interface. Such code might be required to support the replacement of DLLs or the protective transformations, and may come in two types. The first is functions that must have special functionality. For instance, all Naccio transformed programs will require wrapper code for the `LoadLibrary` function to ensure that wrapper DLLs are loaded appropriately. If the platform interface itself does not include a wrapper for `LoadLibrary`, then the application of the standard wrapper is as simple as linking with a library that contains it. If, however, a wrapper is required for `LoadLibrary` both for the protection of resources, and for the standard functionality required by Naccio itself, then the code for these two wrappers must be combined. Automatically combining the two functions into one would be complicated, so standard wrappers are provided to platform interface authors as source code in the same format as the platform interface. If they are not modified then they will simply remain in the platform interface file unchanged. Otherwise, the platform author can choose either to modify the

standard wrapper, or to write a new wrapper that then calls the standard wrapper instead of the true API function. The latter method is best because it makes it easier for new standard wrappers to be provided and added to the platform interface.

This mechanism is sufficient since the number of functions requiring standard wrappers should be minimal. This is not the case of standard code at the entry or exit points of wrapped functions that might be required by the protective transformations. Such code could be required to unlock protected regions of memory needed by the Win32 API as part of the memory protection described in Section 5.2, or to perform stack relocation as part of the more advanced protective transformations suggested in Section 5.2.2. In this case the platform compiler automatically combines this functionality with that of the wrappers generated from the platform interface. This combination only requires the insertion of code at function entry and exit points, and is thus easily performed either through rewriting of source code, or by direct editing of instructions.

Once all the replacement DLLs have been produced, they make up the policy-enforcing system library that is the key output of the policy compiler. They must be stored where they can be loaded by transformed applications, and named so that they can be found and associated with the correct policy. The platform compiler also produces a policy description file used by the application transformer. This file contains the location of the replacement DLLs, and lists of which DLLs should be considered a part of the platform library. It can also list DLLs should be allowed to remain untransformed because they were determined by the platform interface author not to affect protected resources.

### **4.3 Application Transformation**

Once the resource and policy compilers have been run for a given policy, all that remains is to modify an application so that it uses the policy-enforcing library and cannot bypass or alter safety checks. To do this, Naccio/Win32 alters import tables so the application will use the policy-enforcing DLLs. Since the methods of linking to DLLs are simple and processor-independent, wrappers can be applied efficiently and quickly. Most of the effort required of the application transformer is in performing the protective transformations

described in Chapter 5.

Before any transformation can take place, though, the application transformer must identify the code that must be modified. The application transformer is run by providing the name of the application executable, and optionally of any additional DLLs that should be included as part of the application. The transformer then examines the import tables of these files to find which DLLs they link implicitly, and continues this process recursively on those DLLs to produce a complete list of DLLs linked implicitly by the application. The name of each system DLL used by the application is replaced by the corresponding policy-enforcing DLL. The application executable and all DLLs it uses (except system DLLs) are considered untrusted code and are transformed accordingly. This means that any libraries (such as the C runtime library) that were statically linked into the application executable will be transformed as well.

Applications are transformed to use the policy-enforcing DLLs when linking implicitly simply by replacing all references to Win32 API DLLs with references to corresponding wrapper DLLs produced by the policy compiler. This is performed by simply changing an ASCII string in the import table. After any import table modification, the executable or DLL must be re-bound (using the BIND utility or the `BindImage` function as described in Section 3.1) to ensure that function entry point addresses are updated, since those in the replacement DLL may not be identical to those in the original DLL.

It is not, in general, possible to statically determine all of the DLLs used by an application because of explicit DLL linking. Explicitly linked DLLs can be manually specified to the transformer, or handled by the standard wrapper on the `LoadLibrary` function. That wrapper substitutes policy-enforcing versions of any explicitly linked system DLLs at run-time. Explicitly linked application DLLs that remain untransformed either result in a violation, or in the invocation of the transformer at run-time.

The redirection of DLL linking is simple and efficient, and can be performed independently of the protective transformations. Once both have been applied, the application transformer simply outputs the modified executable and DLLs. The user can run the transformed application normally, and should notice no difference until the first time a policy violation occurs (and data is saved from harm by the Naccio protections).

# Chapter 5

## Protective Transformations

Protective transformations are necessary to prevent an untrusted program from bypassing or compromising the policy-enforcing platform library. The wrappers provide protection based on the use of that library, but cannot provide protection if the proper calling conventions of that library are not enforced, or if the program modifies system resources in ways that do not use the policy-enforcing library. The protective transformations must guarantee that such behavior is not possible.

The key categories of program behavior that must be avoided in order to ensure this:

1. **Direct Kernel or LPC Calls:** The DLLs that implement the Win32 API do so (in the case of Windows NT) either by making calls to the NT Kernel, or by passing local procedure call (LPC) messages to the Win32 protected subsystem. If an untrusted program is allowed to use either of these techniques in its own code, it can bypass the policy-enforcing API calls and manipulate resources directly.
2. **Calls to Unmodified DLLs:** Even if appropriate API calls and link information are modified to replace API DLLs with their policy-enforcing version, the original version of each DLL is still usually loaded into the application's address space so that it can be used by the wrappers themselves. If untrusted code can determine or guess the location at which the DLL is loaded and make calls directly to it, then it could bypass the wrapper-based protections.
3. **Self-Modifying Code:** All Naccio protections are based on the examination and

transformation of code before run-time. If code could be modified at run-time, then the protections can be bypassed. Dangerous self-modifying code includes the modification of code already transformed (or code included in safety checking), as well as the construction of new code that is then executed.

4. **Modification of Policy State:** Most safety policies maintain some state (such as the number of bytes written to a file) that they update appropriately and use to perform safety checks. Untrusted code must be prevented from directly modifying this data.
5. **Modification of Win32 API Private Data:** Some of the Win32 API functions themselves maintain data in the application's memory, which while it is safe from the point of view of the operating system, could be modified in such a way as to alter the behavior of the API functions and thus invalidate the platform interface. In such cases, this data must also be protected from untrusted code.
6. **Modification of Local Data of Trusted Code:** In a multithreaded program, it could be the case that trusted code (such as safety checking and API functions) is running in parallel with untrusted code in another thread. The local data used by the trusted code is stored on its thread's stack, located in the application's memory. If untrusted code is allowed to modify this data, it could alter the behavior of trusted code in ways that could compromise checking, allowing undetected policy violations.

Each of these behaviors is dangerous from the standpoint of policy enforcement. Each is also rarely present in well-behaved programs. Properly written programs should perform all system functions by using the appropriate API calls, and should not attempt to modify data not specifically allocated to them. Self-modifying code is rarely used by non-malicious programs. Any attempt to control the above behaviors in a way that allowed them in some cases but not in others, rather than disallowing them entirely, would likely be very complicated and error-prone, and have a correspondingly high cost in robustness and performance. Thus, it seems reasonable to completely disallow all of the behaviors above, at least in cases that cannot be easily determined to be non-dangerous. The number of well-behaved applications that cannot be supported as a result is expected to be small.

Direct kernel and LPC calls can be disallowed through static analysis. During the protective transformation phase, the untrusted executable is scanned for code that makes such calls. Kernel calls are easy to detect, since they require special instructions to make a trap to the system kernel. Kernel traps that can be guaranteed to be safe can be allowed to remain, but any others will be replaced by instructions that cause a violation to be raised. LPC calls are more complicated, but can still be detected and handled in the same way.

The other listed behaviors can be separated into two categories, and the enforcement mechanisms used to avoid them similarly separated into two parts. If Naccio is able to monitor and control the code that may be executed, then it can prevent direct calls to illegal DLLs, or the construction and execution of arbitrary code in data areas. Modification of already existing code is a case of the modification of restricted data, as are the remainder of the listed behaviors. The rest of this chapter is divided between the two categories of enforcement mechanisms that make up the majority of the protective transformations: control-flow and memory safety. The current prototype implementation of Naccio/Win32 does not include the protective transformations, so some aspects of this design are as yet untested.

## 5.1 Control-Flow Safety

In the presence of code such as indirect jumps<sup>1</sup>, the control-flow of a program cannot always be completely determined statically. Thus, run-time checks are needed to avoid illegal control-flow behavior. Naccio makes use of Software-Based Fault Isolation (SFI) [37], a technique that limits control-flow and memory access by transforming code. Details of SFI are described in Section 5.1.1.

Naccio/Win32 uses a limited form of SFI that transforms only jump instructions (alternate mechanisms used to control memory access are described in Section 5.2). These transformations limit the legal targets of any transformed jump to a single region of memory. The way that this is accomplished is described in Section 5.1.2. Naccio/Win32 uses

---

<sup>1</sup>Instructions that transfer control to a location determined at run-time based on the contents of a register or memory location. Such instructions are often used as part of the compilation of procedure calls and switch statements.

this functionality to limit the control-flow of the code in any executable or DLL to its code segment, which is produced as a single region by nearly all Win32 compilers.<sup>2</sup>

This prevents the application from constructing code outside of its code segment, but Naccio must still prevent the application from modifying code within its code segment. To avoid this, the code segment must remain read-only. This is accomplished using virtual memory protection.<sup>3</sup> Since applications are normally able to freely modify the virtual memory protection on their code segment, untrusted code must be prevented from doing so by wrappers on virtual memory routines, as part of the memory safety described in Section 5.2.

Once control-flow has been restricted to an executable or DLL's code segment the only other difficulty is to allow control to leave that code segment at valid times, such as calls to legal DLL functions. The method by which this is allowed is described in Section 5.1.3, after the details of the control-flow limitations.

### 5.1.1 SFI Background<sup>4</sup>

Software-Based Fault Isolation transforms a program to limit the regions of memory that it can read and write, and from which it can execute instructions. SFI imposes its limitations by replacing the three basic types of instructions that access memory – write, read (optionally), and jump instructions – with sequences of instructions to perform checking on the target address before performing the write, read, or jump. Only those instructions whose address cannot be determined at transform-time must be protected with run-time checks so that the address can be checked at run-time after it is calculated. Since the three types of instructions are transformed separately, the memory segments to which they are limited can be different, and often are. SFI usually presupposes that code and data segments are separate so that all addresses in the code segment can be assumed to correspond to legal, transformed instructions.

---

<sup>2</sup>If an application has multiple code segments, they must be combined by the transformer to avoid the difficulties of checking control-flow between different code segments.

<sup>3</sup>The Win32 API does support separate execute permissions, but most processors make no distinction between read and execute access, so this feature cannot be utilized.

<sup>4</sup>Much of the detail here is based on [37], [29], and [30], which include more complete details of SFI.

The two most common forms that the SFI checking instructions take are segment matching and sandboxing. Segment matching performs fault detection by comparing the high bits of the address in use to those of the legal memory segment, and branching to a special violation routine if needed. Sandboxing simply uses bitmasks to set the high bits of the memory address to those of the legal segment. As opposed to fault detection, this is known as fault avoidance. If transformed code attempts to access an illegal address it will not result in any visible violation. Instead, the access will be redirected to some arbitrary address within the legal segment. This could cause the code to have unpredictable behavior, but this behavior is still guaranteed not to access illegal memory. Sandboxing is a more efficient solution because the sequence of instructions that it uses is shorter.

Since every instruction included in the checking increases the overhead introduced, optimizations that reduce the number of inserted instructions are desirable. A notable example is the use of guard zones. Many processors include instructions that access memory using the value of a register combined with a fixed offset, typically with a small possible range of values. For the target address to be checked, this offset must be added to the value in the register before checking is attempted. This added computation can, however, be avoided if the legal memory segment is surrounded by guarded zones of memory as large as the possible value of the fixed offset. Using the operating system's virtual memory protection, these zones can be made to cause hardware address faults any time an address within them is accessed. That fault can be detected by the trusted container application and reported as a violation. Once these guard zones are in place, checking instructions must only test the contents of the register itself, trusting that no fixed offset could make a legal address in the register into an address beyond the guard zones.

An important aspect of SFI is the need for dedicated registers. A typical SFI system needs approximately five registers that cannot be used in any untrusted code. The reason for this is the ability of untrusted code to jump to the final instruction in the sequence of instructions created by SFI, bypassing the checking instructions before it. The standard solution to this is for the final instruction of the checking sequence to use a specially dedicated register. Since untrusted code cannot use this register directly it is guaranteed to always contain a legal address, and jumps past the checking instructions thus cannot allow



access to illegal addresses. One dedicated register is needed for each memory segment in use, and additional dedicated registers are needed to store the segment address itself, and the bitmasks used to examine or clear specified bits in checking instructions. The use of dedicated registers has been shown to have only a small performance penalty on processors with many general-purpose registers. On those with few, including the Intel x86, it is likely that dedication of five registers would be prohibitively expensive.

An alternative to the dedicated registers of traditional SFI is to limit jump instructions not simply to a single memory segment, but to known valid targets. All that is required to eliminate the need for dedicated registers is to ensure that the SFI address-checking instructions are not legal targets, but additional restrictions can also be applied. This was done by MiSFIT [30], which redirected all jump instructions to a specialized routine that compared their target against a list of known valid target addresses. The valid target addresses were calculated as part of code generation, and augmented by new function addresses introduced by dynamic linking. MiSFIT's control-flow safety has significantly higher overhead than that of traditional SFI, but far lower overhead than that which would be introduced by five dedicated registers on a register-poor processor such as the x86.

Once SFI has been used to limit control-flow to a single code segment, the remaining difficulty is allowing it to pass out of that code segment in legal ways. The method usually used to do this is called remote procedure call (RPC). To make an RPC call the untrusted code sets up argument values in registers or memory, and jumps to one of several RPC stubs within its code segment. The stub contains a jump instruction using a fixed address, which allows control-flow to pass out of the code segment to an appropriate RPC routine. The RPC routine checks argument values, and possibly copies data from the untrusted code's memory into other regions. When the RPC call is complete, the RPC routine must copy return-values back, and check that dedicated registers are in legal states before jumping back to the return address, which must also be checked to be within the untrusted code's legal code segment.

### 5.1.2 Use of SFI in Naccio/Win32

The Naccio/Win32 protective transformations use a variant of SFI that differs from traditional SFI in several ways. The differences exist purely as performance enhancements, since standard SFI can provide all the required guarantees. First, since memory protection in Naccio/Win32 is handled by separate memory protection mechanisms (described in Section 5.2) SFI is only applied to jump instructions. The lack of the need to apply SFI to memory access instructions also allows for an optimization that eliminates the need for dedicated registers.

The alteration is to replace indirect jump instructions by fixed jumps to specialized routines, in a way very similar to normal RPC mechanisms. These routines are located outside of the code segment, but do not contain any code beyond the normal SFI checking (either fault detection or fault avoidance). The difference is that rather than utilizing a dedicated register, the target address is maintained in the original register. The reason that this can be done safely is that these routines are located outside of the code segment. Thus there is no danger of untrusted code successfully jumping directly to the final jump instruction, and thus there need be no assurance that the register used as the target contains a valid address at all times.

One such routine is provided for each possible target register (differing also based on whether that register holds the address, or a pointer to the address in memory), and jump instructions are rewritten accordingly. Indirect jump instructions that utilize a fixed offset can be handled by the use of guard zones as described in the previous section, and the load address of the executable's code segment can be adjusted appropriately to ensure that sufficient space for the guard pages is available. Windows' virtual memory system provides the ability to create guarded pages, access to which causes an exception, which will result in the running of a violation routine defined by Naccio.

The checking performed by these routines can usually be the same as that of standard SFI (either sandboxing or segment matching). Some programs may require some special checking on the x86 platform. This is because unlike most platforms to which SFI has previously been applied, x86 has variable-length instructions that need not be aligned

on 32-bit boundaries. Thus it is possible for the same set of bytes to be interpreted as either a constant value included in a previous instruction, or as the start of an instruction, depending on the jump that reached those bytes. If that embedded instruction is a potentially dangerous one, such as an indirect jump or the halt instruction, then jumps to it must not be allowed<sup>5</sup>. This contradicts the normal assumption that all code in the code segment is legal. The dangerous instruction cannot be transformed without invalidating its use as a constant in the preceding instruction. Cases where dangerous instructions are hidden in this way can, however, be detected statically. Programs with no such hidden instructions can be transformed to use normal SFI checking. Those which do have such instructions must perform some extra checking. MiSFIT's checking based on a list of legal addresses avoids the problem, but requires significant cost for building and checking the list. In the likely case that only a very small number of hidden dangerous instructions exist, it will be much more efficient to perform normal SFI checking, and then explicitly check each address against a list of illegal addresses that correspond to hidden instructions. In cases where there are many such instructions, MiSFIT-like checking can be performed, or no-op instructions can be inserted or other transformations performed to avoid use of the hidden instruction. It is believed that such cases will be very rare, since the number instructions considered dangerous is very limited.

Naccio's optimized form of SFI is suitable only for jump instructions due to performance concerns. In standard SFI, an indirect jump instruction is replaced by some number of checking instructions, followed by a similar indirect jump instruction. In this system it is replaced by a jump to a fixed address, which will be followed at run-time by the same checking instructions and eventual indirect jump. The address can be fixed, so it will not cause any additional penalty on modern processors such as pipeline stalls or branch-prediction failures. The total overhead is thus likely to be only one instruction-cycle (though instruction cache performance could be affected as well). On x86 in particular, this overhead is significantly less than the likely overhead of dedicating one or more registers.

---

<sup>5</sup>As with other dangerous behavior such as self-modifying code, Naccio/Win32 makes no attempt to support programs that attempt to use dangerous instructions hidden in this way. The class of programs disallowed by this is likely to be very small indeed.

An additional benefit is that since on most architectures direct and indirect jump instructions have equal width, no instruction relocations will be required to replace the instructions. In fact, the increase in code size is fixed to the size of the address-checking routines, rather than variable due to the insertion of instructions at every indirect jump.

Applying this method of SFI to memory instructions would be inappropriate for several reasons. First, indirect memory instructions (i.e. pointer access) are much more frequent than indirect jumps, so the overhead of the extra instruction would be less effectively counterbalanced by the elimination of the fixed overhead of dedicated registers. Additionally, even RISC architectures often have several types of indirect memory instructions (such as 8-bit, 16-bit, and 32-bit versions), requiring a larger number of address-checking routines to be inserted. On CISC architectures such as the x86, where memory access can be performed as a portion of more complicated instructions (such as arithmetic operations or multi-byte copies), there are even more possibilities to handle.

Applying this optimization to jump instructions, though, simplifies the application of SFI to Naccio/Win32 and reduces the associated performance penalty. The remaining details are identical to standard SFI implementations, and provide the same functionality, as required for the overall strategy of control-flow safety in the protective transformations.

### **5.1.3 Non-Local Control-Flow**

SFI forces jumps to remain within a single code segment, thus eliminating illegal DLL calls, and self-modifying code (since the code segment will be enforceably read-only). Like any SFI system Naccio/Win32 must have some method of allowing legal transfers of control out of the code segment. DLL calls represent the only legal jumps out of the code region, and must receive special handling. Jumps to implicitly linked DLLs will all be in the form of indirect jumps via addresses stored in the IAT (described in Section 3.1), and are thus easy to recognize. The IAT resides in read-only memory (enforceable by Naccio memory protection) so it cannot be modified after the loader runs. Thus the IAT can contain only valid DLL entry points based on the import and export tables already adjusted by the policy transformations, and thus implicitly linked DLL calls are known to be safe without further

modifications.

Explicitly linked DLLs, and the associated jumps to procedure pointers returned by `GetProcAddress`, are a far more difficult problem. This can be accomplished by applying a wrapper to `GetProcAddress` that returns the address of a pre-prepared stub routine inside the code segment that dispatches to the appropriate point outside the code segment. A different stub can be used for each explicitly-linked function, and the wrapper to `GetProcAddress` must simply cause the stub code to jump to the appropriate location by storing the DLL address in a protected memory location used by the jump in the stub. This essentially creates a dynamic IAT for explicitly linked DLLs, which the `GetProcAddress` wrapper is capable of modifying because it is considered trusted code by the Naccio/Win32 memory protections.

The number of stubs needed is equal to the maximal number of explicitly-linked functions active at once. If the number of explicitly linked functions at run-time exceeded the number of stubs, an error would result. Since stubs are small (only a few instructions) it is possible to create many of them, with the actual number set at transform-time. Stubs can be re-used if the DLL containing the corresponding function is unloaded. In fact, stubs must be either re-used or cleared in this situation, lest the remaining stub be used to jump to a no longer legal memory address.

Returns from DLL functions are also a valid jump out of a code segment. Since wrapper DLLs are not transformed by the protective transformations, the only relevant DLL returns are those from DLL functions in application DLLs. On processors that use dedicated return instructions, these can be handled by another address-checking routine outside the code segment. It would differ from others in that it would consider addresses in any application code segment to be legal (a list of such segments can be provided by the application transformer for implicitly linked DLLs, and extended by the `LoadLibrary` wrapper for explicitly linked DLLs). If the processor has no dedicated return instruction, then any indirect jump instruction could theoretically be used for a DLL return. In this case, the checking of alternate code segments must be performed for any target address that is found to be outside of the active code segment. This introduces no additional overhead for DLL returns (since the current code segment would have to be checked anyway to handle

internal procedure returns), and for other instructions it only requires that fault detection, rather than fault avoidance be used, so that “illegal” addresses result in a branch that can lead to further checking, rather than simply to an incorrect address somewhere in the legal code segment.

## 5.2 Memory Safety

Naccio’s protective transformations must protect many types of data in memory, including policy state, the state of API routines themselves, code segments of both trusted and untrusted code, and the local data of threads running trusted code. Since an application normally can read and write all of its own memory, Naccio/Win32 must make certain regions of memory non-writable by untrusted code, but writable to trusted code. Reading such memory could be disallowed as well, but it is unlikely that the risks introduced by reads are significant enough to justify the additional cost and complexity.

One solution would be to apply full SFI to all untrusted code in order to control their memory writes as well as their control-flow. Since the memory access of untrusted code must be limited by the protection of several regions, rather than by limitation to a single region, the SFI method applied would be more complicated than those explored in most previous work, and would be likely to have a higher performance overhead.

Instead, Naccio/Win32 takes advantage of the Windows operating system’s virtual memory protection features and the existing wrapper mechanism to produce a more efficient solution. It uses virtual memory protection to mark the guarded regions of memory as read-only, and uses wrappers to enforce that access restriction. This can be done simply and efficiently, and in a way that depends only on the Win32 API, unlike the processor-specific SFI transformations. The primary shortcoming of this solution is that it can provide guaranteed protection only in a single-threaded program. Section 5.2.1 presents this solution, while Section 5.2.2 discusses the issues raised by multithreading and describes possible systems for multithreaded memory safety in Naccio/Win32. Multithreaded memory safety for Win32 is considered an open issue, and thus no particular solution is included in the current Naccio/Win32 design. Instead, several possible solutions are presented, and future

investigation will reveal which can provide the best protection at the lowest cost.

### **5.2.1 Single-Thread Memory Protection**

In an environment with only a single thread of control within a process, Naccio's memory safety can be based entirely on virtual memory protection and specialized wrappers on memory-access routines, with no direct transformations of user code required. Win32 provides page-based memory protection, but allows applications to freely change the protection level on their own memory. Thus, except in the case of shared resources (on which permissions are enforced), memory access permissions act only as a safeguard against program bugs, rather than an impediment to a hostile program. Since all modifications to memory protections must be performed through the use of API calls, however, Naccio can control an untrusted program's ability to change the permissions on its memory. This method could be used to enforce read-only access on memory pages containing data that should not be changed by untrusted code, by raising violations if an untrusted program attempts to enable write access to those pages. Read access could also be disallowed by this method if desired.

Since code generated by Naccio can make calls to API routines without going through wrappers, it can freely enable write access when necessary. Vital memory regions can thus be kept read-only when user code executes, and changed temporarily to read-write when API or checking code needs to modify data. This requires frequent calls to the API routines that change memory protection, which have an associated overhead due to the kernel call involved. If the locking and unlocking of memory access is properly placed, this overhead should not be needed at all for read access, but only when writes must be performed. Since reads and writes can be performed by policy-specific code, the placement of these locking and unlocking calls is performed by an automated analysis based on the analysis data provided by the resource compiler.

Unlocking and relocking of memory used by the API functions themselves must be performed by wrappers. The code to do so is automatically inserted at the entry and exit points of wrapper code as described in Section 4.2 with no need for intervention by the author of the platform interface. API functions that could previously be passed through

with no wrapper may require a wrapper for the purpose of this form of memory protection.

The identification of memory to be protected can be done relatively easily. All code segments must be protected. All data used by resource checks can be stored in a single heap and that heap's region of memory protected. In some cases, data used by the API itself can also be stored in that heap by changing the default heap location before API functions are called. Some API functions may use writable regions of DLL code, and those must also be detected and protected. The existence of these regions can be determined by examination of API documentation and the DLL files themselves, and is a part of the supplemental data provided in the platform interface.

The reason that this solution cannot support multiple threads is that it is time-based (requiring no untrusted code to run when protected memory is unlocked) and provides no protection for local data on the stack (since it is not vulnerable except when trusted and untrusted code run simultaneously). The issues surrounding multithreading and memory protection in Naccio are currently under investigation, and described in the next section. The single threaded solution, however, is applicable in many situations. Its primary advantage is its simplicity and its independence of the underlying processor type. Many Win32 applications use only a single thread, particularly command-line based application. If Naccio/Win32 or a similar system were to be applied to helper applications such as applets downloaded from the Web, then limiting code to a single thread would be a reasonable way to save complexity and performance. This single thread limitation can be enforced by wrappers on the API functions that create new threads. In addition, while Win32 is a thread-based architecture, many modern operating systems (including most UNIX variants) use a process-based architecture where multithreading would not be an issue. It is likely that a simple variant of this single-thread memory protection system would be sufficient for Naccio implementations for those operating system platforms.

## **5.2.2 Multithreading Issues**

Multithreading in Win32 allows multiple threads of control to exist within a single application. The threads are scheduled separately, but share a single address space. This makes



sharing information between threads efficient, but foregoes some of the additional protections that can be provided by the operating system between processes. For cooperative and well-behaved threads, which make proper use of synchronization primitives to avoid race conditions, the performance and simplicity benefits of threads often outweigh the dangers. In Naccio/Win32, however, trusted or untrusted code, and protected or unprotected data can be used by different threads simultaneously. This presents a number of potential race-condition based vulnerabilities:

- The locking and unlocking mechanisms used by the memory protection system described in the previous section relies on only trusted code being able to access protected memory while it is unlocked. The ability to write to protected memory must be unlocked before trusted code performs a write, and relocked before untrusted code is run again. If untrusted code is running simultaneously, it may be able to modify a piece of protected memory while it is unlocked on behalf of trusted code.
- The local data of a thread, stored on its stack (one of which is allocated for each thread) is not protected at all by the single-thread memory protection scheme. Since alteration of this data could affect the execution of trusted code, untrusted code cannot be allowed to modify it while trusted code is running.
- Threads are not protected from each other in any way. This includes the ability of one thread to modify another thread's context, including its program counter, stack pointer, and register values. Unchecked usage of these capabilities could allow untrusted code to modify the behavior of trusted code in another thread, or to cause illegal code to be executed by another thread.

The last of these issues can be easily dealt with by wrappers applied to Win32 API functions. The other vulnerabilities must be avoided by the protective transformations. The rest of this section describes possible methods for extending the capabilities of the Naccio/Win32 memory safety mechanisms to provide protection for multithreaded applications. Not all of the suggested solutions provide all of the necessary protections when used alone, but a combination of them would create a sufficiently secure system. There has

not as yet been sufficient investigation or testing to know which of these solutions would be best, or which (if any) would be capable of providing the required protections with an acceptable performance loss. An attempt is made in the descriptions below to predict the benefits and pitfalls of each approach. SFI is the most promising approach, particularly because application of more thorough SFI would be easy to integrate with the application of control-flow SFI already performed by the application transformer.

**Halting of untrusted threads** would allow the time-based protections of the single-thread memory safety scheme to be effective in a multithreaded environment. Each time trusted code needs to have access to protected memory, it could halt all other threads to ensure that it is the only code executing in the process. This could easily be accomplished through use of the Win32 API, though care would have to be taken to ensure that arguments on the stack could not be modified in a way that would cause the API calls to be ineffective.

While this solution would provide the required protection, it would limit the program to a single thread whenever trusted code is modifying protected resources or performing computations that would be susceptible to stack modification. If this included time spent in long API functions performing I/O, then one of the primary benefits of multithreading would be lost. In addition, care would have to be taken when applying this method to avoid deadlock situations. For example, halting all other threads before a call to an API routine that waited for the release of a synchronization object could cause the program to halt indefinitely if that object was held by a halted thread.

**SFI** could be applied to memory write instructions as well as jumps to enforce the required memory restrictions. Since the limitations introduced are not as simple as a single legal memory segment, the length of the sequence of checking instructions (and thus the overhead introduced) would be higher than in standard SFI. In addition, as described in Section 5.1, the optimized version of SFI used for control-flow safety is not well-suited to memory protection. Thus providing write-based SFI in Naccio/Win32 would require either the use of dedicated registers, or the use of specialized control-flow checking sequences (such as those used by MiSFIT) that avoid the problems inherent in unrestricted jumps within the code segment.

Using SFI to protect local data in thread stacks is also possible if the checking code is able to retrieve data about the current thread (available through the Win32 API). Checking sequences can then consider the space allocated for the current thread's stack as a legal memory segment, but all other thread stacks as illegal. Since this could reject some valid programs that pass pointers to local data for temporary use by another thread, the checking code could instead keep track of which threads are or are not executing trusted code, and prohibit access to only the stacks of trusted threads. This would require some additional work both by SFI checking sequences and by wrappers to keep track of trusted and untrusted threads, but the additional overhead could potentially be much lower than that introduced by other memory protection schemes.

**Shared memory** could be used to hold protected data, in order to take advantage of the memory protection Windows enforces on shared memory regions. Protected data could be stored in a shared segment mapped read-only into the application's memory space. Another watchdog process, started with the application, would be able to write to that segment. This solution allows reads to protected state to proceed with no additional overhead, while writes to the segment would have to be performed via LPC messages passed to the watchdog process. That LPC would introduce additional costs, but not much more than that introduced by the unlocking and relocking used by the single-thread scheme.

The disadvantage of this scheme is that it provides no protection for thread stacks, and may not be able to protect writable data in system DLLs if that data cannot be relocated into the shared segment. The former problem can be avoided by moving portions of checking code into the watchdog process. If all safety checks that would be susceptible to modifications of local data are performed outside the application process, then no interference is possible. The inherent trade-off is that the process boundary must be crossed in cases even when writes to the shared segment may not be required, and each crossing has an associated cost. The severity of the problem presented by local DLL data cannot be known without a more thorough inspection of the Win32 API DLLs. If such data must be protected, it may be necessary to modify the system DLLs directly rather than simply using wrapper DLLs to intercept system calls.

**Protected stacks** can be used to reduce the difficulties introduced by stack data when applying the schemes above. Rather than attempting to protect the stack data of threads at all times, each thread could be allocated a region of protected memory (controlled by SFI or some other method) to hold its stack when it is executing trusted code. At the entry point of any trusted code that would be vulnerable to local data manipulation, the stack pointer of the executing thread will be moved from the normal stack region to the protected region. After the trusted code has completed running the pointer would be moved back. This can be accomplished by code inserted at the entry and exit point of every relevant wrapper function.

Trusted code written to take this into account can run normally using the protected stack. If code not aware of the stack change must be run, any arguments passed on the stack will need to be copied into this secure region so that they can be properly accessed. Any modified data on the stack that will be visible to the callee (such as return values) must be copied back to the original stack after the trusted code is complete. Copying must include enough of the stack that any stack-relative accesses will be properly handled.

Additional copying may be needed in cases where non-stack data must be used by trusted code in a way not tolerant of modifications. For instance, if checking must be performed on the contents of a structure passed by pointer to an API call, then the entire structure must be copied into protected memory so that its contents cannot be changed between the time of the checking and the time of the API call. This process is similar to the copying of arguments in RPC and kernel calls. The amount of copying required will be heavily dependent on the API call and resources in question, so the platform interface author will specify how memory objects must be copied to and from protected memory to provide proper protection. Copying all data passed would be inefficient, or even impossible when pointers are passed without knowledge of the amount of memory they may reference. The platform interface author will know what data the API call must use, and whether resource operations are dependent on that data, and can thus make the correct decision to perform necessary copying without wasting effort on data that can remain in unprotected memory. It is unknown whether the copying overhead of this approach would outweigh its benefits when compared with SFI-based protection of thread stacks.

# Chapter 6

## Prototype Implementation

The preceding chapters have presented the design of Naccio/Win32, but leave many details unspecified. This chapter describes the current prototype implementation of Naccio/Win32, which has been successfully tested on Windows NT on the x86 and Alpha platforms using sample programs, including those described in Chapter 7. The primary purpose of this prototype is as a proof of concept, and a tool for generating experimental results for the investigation of the practicality of Naccio/Win32, as well as driving future work on Naccio. Ease of implementation was a priority, thus the absolute performance of this implementation is less than what could reasonably be accomplished. It is likely, however, that it displays similar qualitative characteristics to a more complete implementation, and is thus a useful tool for analysis of the soundness and usefulness of the Naccio/Win32 design.

The choices and details in the implementation of each of the major components of Naccio/Win32 are described in the following sections. Limitations of the prototype implementation, which does not fully implement all of the features of the Naccio/Win32 design, are described as well.

### 6.1 The Resource Compiler

The platform-independent portion of the Naccio/Win32 prototype is the same as that used in the Naccio/JavaVM prototype described in [7]. The system is written in Java to allow it to be run on any platform. The mechanisms for parsing, analyzing, and optimizing resource

descriptions and safety policies are thus complete, and the only meaningful difference introduced by Naccio/Win32 is the back-end that generates a resource implementation suitable for the Windows platform.

This implementation is C source code, separated into a header file (`RESOURCE . H`) and a source file (`RESOURCE . C`). Together they are compiled to a single DLL (`RESOURCE . DLL`) containing entry points for all resource operations. The header file is then included by all platform interface source files.

The header file contains type declarations for all resource state in the form of structures, as well as function declarations for all resource operations and the compiler directives to cause Microsoft Visual C++ to compile them as DLL exports. Function names follow the conventions introduced in Section 4.1. The inclusion of this header by platform interface sources allows for easy elimination of unnecessary calls to resource operations. When a resource operation has been determined to have no effect, the resource compiler does not produce a function declaration, but instead generates a macro definition with an empty body. As a result, the preprocessor automatically eliminates calls to the resource operation as platform interface sources are compiled.

The C file produced by the resource compiler contains implementations of resource operations that perform meaningful work. Since the language used to define safety checks is similar to C in its use of operators and functions (it is based on Java), the generation of C code is straightforward. The Naccio library is implemented as a DLL and provides most of the nontrivial functions called by checking code. The two areas in which nontrivial translation is needed are resource objects and strings. Handling of resource objects is merely a matter of syntax. Safety checking code uses Java-like conventions for access to the state variables and operations of resources. Translating this to valid C requires only replacing operators and automatically passing resource objects as the first argument to resource operations.

The `String` data type utilized by safety checking code does not correspond to any primitive C type. Safety checking treats a `String` as an immutable object, and performs no explicit memory allocation or deallocation. A `String` is translated into a pointer to a structure that contains a pointer to a character buffer, but also information about the size of

that buffer, and whether the buffer is read-only (usually a string literal) or in heap-allocated read/write memory. The reason for this is that the implementation data type is mutable, and the implementation of some string operations takes advantage of this. The implementation of resources also includes manual allocation and deallocation of `Strings` as part of the translation process. The two available operations on a `String` in safety checking code are assignment and concatenation, and both are implemented by special library functions called by translated code. Temporary string variables used by these functions are allocated only once at the beginning of a resource operation, and re-used to avoid unnecessary memory allocation.

The only additional resource operations generated by the resource compiler prototype are the allocation operation (which includes safety checking code associated with the constructor) and the deallocation operation (which includes code associated with the `finalize` operation). These perform explicit allocation and deallocation, without any form of automatic memory management such as garbage collection or reference counting. The platform interface must ensure that a resource object is deallocated at the proper time. No synchronization operations are generated by the resource compiler, and thus synchronization of access to resource objects by multiple threads is not supported in the current prototype. Due to the limitations of memory protection described in Section 5.2, the current prototype only includes full support for applications with a single thread of control.

## **6.2 The Platform Compiler**

The platform compiler and platform interface format for the Naccio/Win32 prototype were designed to be simple to implement, while still allowing maximal flexibility on the part of the platform-interface author to explore different possibilities. The platform interface files are defined in a simple easily parsed format. They contain the names of functions to be wrapped, disallowed entirely, or allowed unmodified, default behavior to take for unmentioned functions, and the name of a C source file that is to be compiled to produce the wrapper functions. This source file is provided directly by the platform interface author. The prototype implementation does not perform any optimization of wrappers beyond that

```
EXPORTS
DeleteFileA=wrapper_DeleteFileA
MulDiv=KERNEL32.MulDiv
DeleteFileW=NACC_LIB.IllegalAPIViolation
```

Figure 6-1: Excerpt from the file `KERNEL32.DEF` generated by the platform compiler

already provided by the resource header file. Thus there is no parsing or analysis of the platform interface source code, which can simply be compiled by a standard C compiler.

The platform compiler runs the Microsoft Visual C++ compiler to create DLLs from wrapper the source code. The appropriate linker directives are used for forwarding of references to null wrappers, or forwarding of disallowed functions to violation functions. This is performed through the use of a definition file that specifies all function names to be exported. This file is automatically generated by the platform compiler.

Figure 6-1 shows an excerpt from the definition file for the policy-enforcing version of `KERNEL32.DLL`. It includes a single wrapped function (`DeleteFileA`, which corresponds to the version of the function `DeleteFile` that takes the file name as an ASCII string). It also includes a sample of the method of specifying forwarding for a null wrapper (for the `MulDiv` function, which only performs mathematical operations and thus does not effect resources) and for an illegal function (`DeleteFileW`, which is currently disallowed due a lack of support for Unicode (wide-character) strings). The complete definition file includes an entry of one of these three types for every function exported from the original `KERNEL32.DLL`.

Once the replacement DLLs are compiled, they are given the alternate extension `.NPD` so that they can be distinguished from the true system DLLs by the loader<sup>1</sup>. These files must be placed in a directory on the DLL search path so that they can be found. Since only a single filename extension is currently used, and the application must find the proper policy-enforcing DLLs in its search path at run-time, only a single policy can currently be supported. Supporting multiple policies would require using a different extension for each

---

<sup>1</sup>In order to avoid the need to perform relocations of addresses in the executable (which as described in Section 8.3 is a difficult and time-consuming task), it is desirable for the name of the wrapper DLL to have the same length as that of the original DLL, so an alternate extension is used rather than a prefix or suffix.



```

BOOL wrapper_DeleteFileA(LPCTSTR lpFileName) {
    RFile rf = addRFileByName(lpFileName);

    RFileSystem_delete(rf);
    RFileSystem_observeExists(rf);
    releaseRFile(rf);

    return(DeleteFileA(lpFileName));
}

```

Figure 6-2: Platform interface excerpt showing the wrapper code for `DeleteFileA` in `KERNEL32.DLL`. The `A` in the function name indicates that the argument string is ASCII. `BOOL` and `LPCTSTR` are standard Windows data types (boolean, and pointer to constant string, respectively).

policy, and specifying that extension in the policy description file.

## 6.3 The Platform Interface

The Win32 API is extensive, and not all of it is described by the platform interface included in the prototype Naccio/Win32 implementation. The current platform interface includes only wrappers involving file system access, corresponding to the `RFileSystem` and `RFile` resources described in Section 2.2.1. The wrapper code for file-affecting API calls was generated based on the Win32 API documentation included in the MSDN Library [19]. The platform interface must describe all ways in which API functions can affect resources. The creation of a more complete platform interface is thus a matter of significant time and effort, but not of any significant design difficulty.

The wrapper format uses the naming conventions of the resource operations to include all Naccio-specific information. Figure 6-2 shows a sample wrapper, for the Win32 API function to delete a file by name. The `RFile` data type has been declared by the header file generated by the resource compiler. The appropriate `RFileSystem` operations are called: the `delete` operation for the deletion itself, and the `observeExists` operation representing the fact that the function will return an informative error value to the user code if the named file does not exist. Noticing error behavior such as this provides one key

difficulty in creating an accurate platform interface.

The `addRFileByName` and `releaseRFile` functions are defined in a helper DLL (`RFILEMAP.DLL`). `addRFileByName` creates a new `RFile` object corresponding to a file name, or returns an existing one if a resource object corresponding to the same file has already been created elsewhere in the application. `deleteRFile` indicates that this wrapper no longer needs the `RFile` object. The file map will determine whether to deallocate the object, or maintain it because the file is still in use by the application, for instance if the application had opened the file for writing elsewhere. (In this case the deletion of the file by this API function is likely to cause an error, but since error handling is part of the application's behavior, the wrapper does not interfere, and the file map will maintain the `RFile` object until the other handle to the file is closed.)

## 6.4 The Application Transformer

The current Naccio/Win32 implementation includes only the policy transformations described in Chapter 4. The protective transformations described in Chapter 5 are not implemented by the prototype. This means that while the current prototype can successfully enforce policies on most Win32 applications, it would be vulnerable to targeted attacks.

In the absence of protective transformations, the work of the application transformer is simple and fast. It simply examines import tables to find all application DLLs, and then modifies those import tables to redirect DLL access. The policy description file produced by the platform compiler contains a list of policy-enforcing DLLs, and those are substituted for their originals by altering the filename extension. Non-wrapped system DLLs (identified by their location in system directories) can either be allowed unmodified or treated as application code by user choice. In order to avoid overwriting original files and thus allow easy re-transformation, application DLLs are copied and given an alternate extension (`.NTD`) before they are transformed. The application executable must retain its original name to be able to be run identically, but it is backed up to an alternate extension (`.EXB`) before transformation begins.

The last task of the application transformer is to provide information on the identity

of policy-enforcing, application, and unmodified allowable DLLs for use by the standard wrapper on the `LoadLibrary` function. It does this by producing a small source file containing static arrays of the names of DLLs in the three categories, and compiling that file into a DLL. At run-time, the `LoadLibrary` wrapper can use these arrays to classify the names of DLLs that it must load. The purpose of including arrays in a DLL rather than a data file of another sort is to minimize the run-time cost of loading the data. Currently, the `LoadLibrary` wrapper signals a violation if an unknown DLL is loaded rather than attempting to invoke the application transformer at run-time. The user can manually re-transform the application, specifying the DLL from the violation message on the command-line in order to include it in the transformation.

# Chapter 7

## Results and Analysis

The Naccio/Win32 prototype has enforced a range of file system policies on a number of Windows applications. Due to the limited nature of the current platform interface, many large complex applications cannot be supported properly because they use the API in unsupported ways or manipulate resources not included in the current prototype. The prototype has, however, demonstrated its capabilities and performance with applications and policies that fit within the currently (covered) subset of the API. Section 7.1 describes the sample safety policies used to test the Naccio/Win32 prototype, while Section 7.2 describes the sample applications that were transformed. Section 7.3 then presents performance results generated running these samples with the Naccio/Win32 prototype.

### 7.1 Sample Policies

Current testing has been limited to policies that place limits on file system access. These sample properties are not intended to illustrate a comprehensive set of Naccio/Win32 features, but simply as a useful set of examples for demonstration and testing purposes. More realistic policies would combine these file system policies with protection for other resources, such as the network.

The sample policies used to test Naccio/Win32 are described below. Except for the `Null` and `Combined` policies, all policies used for testing included only a single safety property, and thus the property name and its parameters are given here. Full policy descriptions for

these policies are given in Appendix A.

`Null` performs no checking, but demonstrates the overhead required to enforce any policy.

`LimitPath("d:\\legal")` limits file system access to a single directory tree by matching absolute filenames against a specified prefix. All resource operations corresponding to file access require an `RFile` object as an argument, and those objects are constructed only when a file is to be accessed, so the checking can be performed in the constructor for `RFile` resource objects. This minimizes the amount of checking code needed at run-time.

`ReadOnlyDir("d:\\legal\\readonly")` allows read access to a directory tree, but disallows write access to it, again by comparing absolute filenames against the specified prefix. Since a distinction must be made between reading and writing, the constructor-based checking approach of `LimitPath` cannot be used. Instead, checking code is attached to resource operations that correspond to opening a file for writing, or modifying a file in ways other than writing (such as altering attributes). No checking is required on the often-executed operations to write bytes into files, however, since access must already have been checked when the file was opened.

`NoOverwrite` prohibits untrusted applications from modifying or replacing existing files, as described in Section 2.3, but allows new files to be created. As is the case with `ReadOnlyDir`, checking is needed only on operations that open existing files for writing, and those that affect files directly

`LimitBytesWritten(100000000)` (described in Section 2.3) limits the number of bytes that can be written to files to 100 million. The limit is set high for performance testing to avoid violations.

`Combined` instantiates all of the properties described above (including `LimitPath`, `ReadOnlyDir`, `NoOverwrite`, and `LimitBytesWritten` all with the arguments given above) to enforce all of their restrictions at once.

## 7.2 Sample Applications

Naccio/Win32 should be able to enforce policies on any Win32 application. Its current platform interface, however, limits its support to applications that use the subset of the Win32 API than is currently described. Resource manipulations performed through other API functions cannot be constrained by the Naccio/Win32 prototype.

Though we expect Naccio/Win32 will be used primarily on interactive applications, the sample applications chosen here are non-interactive. Run-time performance testing is more meaningful on non-interactive applications, since for interactive applications the delays introduced by user interaction overshadow the overhead of safety checking. The sample applications are also file-intensive, so as to make extensive use of the portion of the Win32 API described by the prototype platform interface. Less file-intensive applications, or those with a higher degree of user interaction, would suffer smaller performance penalties than those measured here.

The sample applications used for testing were:

`treecopy` – a hand-written program that copies an entire directory tree one file at a time, preserving the directory structure. `treecopy` makes heavy use of the file system with nearly no computation, using the `CopyFile` and `CreateDirectory` API functions.

`tc-verbose` – a version of the `treecopy` program that performs the same operations, but prints the name of each file as it is copied. The overhead associated with console I/O makes the balance of computation vs. file I/O in this application more realistic.

`pkzip` (32-bit command-line version 2.50) – a file compression utility from PKWARE Inc. It was run to compress an entire directory tree into an archive, using maximal compression. `pkzip` makes heavy use of the file system by reading and writing bytes from files, but also includes a significant amount of computation.

An alternate version of the `ReadOnlyDir` and `NoOverwrite` properties had to be used when testing this program. This is because `pkzip` requests write access to files (those being added to the archive) on which it never actually performs `write`

operations. This results in policy violations that limit desirable program behavior. In order to allow this idiosyncratic behavior, the checking in these two policies was moved from the `openWrite` and `openAppend` operations into the `write` operation. This sacrificed performance when compared to the more optimized versions of these policies used for other testing.

All sample applications were run on a sample directory tree containing 1438 files in 57 directories, totalling 31MB. Source and target files were all located within the legal path of the `LimitPath` policy, and source files were located within the read-only path of the `ReadOnlyDir` policy. While violation conditions were thoroughly tested, paths and command-line arguments for performance testing were chosen so that no violations would occur, allowing programs to run to completion.

### **7.3 Performance Results**

The sample applications and policies described above were used to test the performance of Naccio/Win32. We report the time to compile a policy and transform an application, as well as the run-time performance of the transformed applications. All performance results were generated on a 500Mhz Pentium III workstation with 256MB SDRAM running Windows NT 4.0 with Service Pack 3. The affected files were stored on a FAT file system on an EIDE drive. No other programs were active on the test system during testing.

Because of the small set of resources and wrapped API functions supported by the prototype implementation, policy compilation time was relatively short, varying little between policies. The resource compiler ran in 2-3 seconds for the sample policies. The platform compiler ran longer, primarily because of the time taken to compile sources into DLLs, but still completed in just over 5 seconds. Since all of the file system API functions are contained in a single DLL, only a few files were produced, and the amount of disk storage required was also small: 90-130KB to store the policy-specific DLLs and policy description file.

Application transformation is very fast, dominated by the time taken to copy files before transformation so as to preserve their original versions. With the absence of the

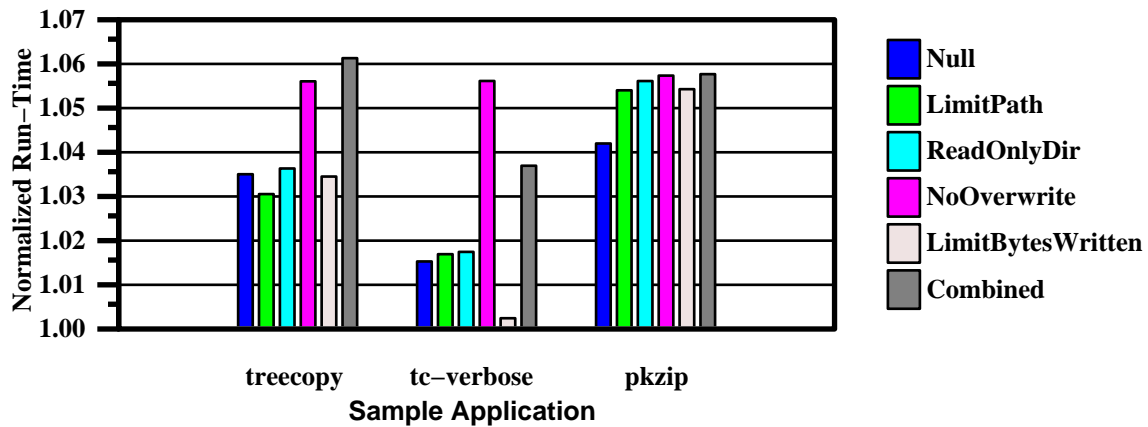


Figure 7-1: Naccio/Win32 Prototype Performance Measurements

protective transformations, the actual modifications made were limited to import tables, and no relocations were required, so the computational overhead of the transformations is negligible. The protective transformations would add additional transformation time, but since the static analyses involved are simple, and much of the work of the memory protection described in Section 5.2 is based on wrappers rather than instruction-level transformation, that time should also be quite small.

Run-time cost is the most important measurement of the performance of the Naccio/Win32 prototype, since it is the aspect of performance most noticeable to a user. Results from enforcing the sample policies on the applications described earlier are shown in Figure 7-1. The run-time of transformed applications is normalized by the run-time of the untransformed application, such that a value of 1.00 would indicate no run-time overhead. All timing results were generated using the average of 100 tests to minimize the effect of transient effects such as buffer and cache performance.

The overhead introduced by Naccio/Win32's safety checks is well within reasonable ranges. Indeed, the change in performance based on transient operating system effects was often much larger than the overhead itself, resulting in run-times up to 50% slower than average even for untransformed code. The highest overhead demonstrated is just over 6% for `treecopy` with the `Combined` policy. These results demonstrate that Naccio/Win32 is capable of enforcing useful policies on real applications without prohibitive performance penalties.



The run-time overhead introduced by Naccio depends on the policy being enforced. The `tc-verbose` demonstrated seemingly anomalous results for the `NoOverwrite` and `LimitBytesWritten` policies, caused by the high and unpredictable overheads associated with console I/O, but the other applications and policies paint a consistent picture of the overhead associated with the sample policies. Since Naccio introduces checking code only where it is necessary to enforce a particular policy, the overhead introduced by transformation itself is small. More aggressive optimizations of wrapper code would eliminate much of the overhead associated with the `Null` policy, and cause a corresponding reduction for other policies. For instance, the wrapper on the API functions to read or write bytes to a file is included even in policies that perform no relevant checking, resulting in the high overhead for the `Null` policy on the `pkzip` application. Optimization of wrappers could eliminate that wrapper entirely in policies that place no constraints on it.

Other policies introduce varying levels of overhead based on their complexity and the frequency of checked API calls. The `LimitPath` policy produces the smallest overhead, since it must perform checking only when a new `RFile` object is constructed. The `LimitBytesWritten` policy, in contrast, must perform checking each time bytes are written to a file, and thus demonstrates higher overhead. The alternate versions of the `ReadOnlyDir` and `NoOverwrite` properties used for `pkzip` require checking on all `write` operations, and thus cause higher overhead for those policies as well as the `Combined` policy.

The time taken by each check is significant as well as the frequency of checking. While the checking performed by the `LimitBytesWritten` policy is frequent, it is also very simple, consisting only of an increment and check of a counter. The `ReadOnlyDir` and `LimitPath` policies must perform string comparisons, making each individual check more expensive. The `NoOverwrite` policy needs to query the existence of files when the `copy` operation is checked, and this query requires a time-consuming API call. Since the `treecopy` and `tc-verbose` applications make heavy use of file copying, they suffer a high penalty from the enforcement of this policy. In all cases except `tc=verbose`, the `Combined` policy imposes the highest run-time overhead, since it combines the checking of all safety properties.

The prototype implementation cannot be used to produce a precise prediction of the performance of an industrial Naccio/Win32 implementation. More extensive checking of a more comprehensive set of resources would incur a higher performance penalty, as would more complicated or application-specific policies.

The protective transformations would also introduce additional run-time overhead that is not reflected in this prototype. Based on previous SFI implementations [37] [29], it is estimated that the jump-only form of SFI described in Section 5.1 would introduce less than 10% performance overhead on average applications, and that the non-local control-flow mechanisms would introduce little additional overhead. The single-thread memory protection mechanism presented in Section 5.2.1 has not been previously tested, so no reliable estimate can be made of the precise performance overhead it would introduce, but intelligent placement of unlocking and relocking operations in the platform interface should allow this overhead to be acceptably small.

Despite these uncertainties, the current Naccio/Win32 prototype is a convincing proof-of-concept. Naccio/Win32 provides a higher degree of flexibility and capability in the description and enforcement of safety policies than perhaps any previously existing code safety system, including interpreted languages and virtual machines, which have been known to run 20 to 50 times slower than compiled C code [30].

# Chapter 8

## Related Work<sup>1</sup>

When they are compared to Naccio, previous code safety efforts can be divided conceptually between low-level and high-level code safety. High-Level code safety corresponds to the resource-based safety policies that most users wish to enforce, and is analogous to Naccio's policy transformations. Low-level code safety provides basic protections on code capabilities and prevents the bypassing of higher level policies, and is analogous to Naccio's protective transformations. The following two sections survey previous work in code safety, dividing coverage between those two categories. Naccio uses code transformation to provide both high-level and low-level code safety.

While there has been significant previous work in code safety, code transformation has been used only rarely for this purpose. The most notable previous use of code transformation for safety is Software-Based Fault Isolation (SFI) [37] [29], described below in Section 8.1. SFI modifies executables at machine-code level to provide memory safety, though more recent work has generalized it to enforce a larger class of policies. SFI is being used to provide non-bypassability of wrappers on Windows NT in the Generic Software Wrappers project [9] described below in Section 8.2. Security Automata SFI Implementation (SASI) [36], a system based on execution monitoring [28], and Ariel [23] (also described in Section 8.2) both enforce safety policies by inserting code into executables in ways similar to Naccio. Most previous work in editing of executables, however, has been directed toward

---

<sup>1</sup>This section is based in part on [7].

other purposes. Section 8.3 describes relevant work in this field.

## 8.1 Low-Level Code Safety

The minimum low-level code safety required to support most high-level code safety mechanisms is control-flow safety, memory safety, and stack safety [15]. Without control-flow safety, an attacker could construct a program that jumps directly to the body of a system call, bypassing any checking code. Without memory safety, a program could modify its own code, and possibly the code and data of other programs. Without stack safety, code in a program could alter the local stack data belonging to itself or another piece of code to cause illegal actions to occur.

**Safe programming languages** can provide the memory safety and memory safety necessary for low-level code safety. Memory safety can be provided 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. Type safe languages also provide control flow safety, since all jumps are through well-defined language control structures. Several languages that guarantee varying degrees of type and memory safety have been designed, including CLU [18], ML [20], and Java [35]. Unfortunately, safe languages are useful in providing code safety 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 directly in the execution environment, but this has serious performance and code-disclosure consequences. This can only be avoided if there is some way to verify that compiled code was created using the safe language. SPIN [3] suggests using cryptographic signatures from a trusted compiler to verify this, allowing trusted extensions to an operating system kernel using code written in a safe subset of Modula-3. The need for a trusted compiler, however, adds bottlenecks and vulnerabilities to the system. Naccio avoids these difficulties by operating on fully compiled executables. Those executables can be generated from any language, avoiding the limitations placed on programmers by systems based on safe languages.

**Verification** can ensure low-level code safety without the need for a safe language by directly verifying necessary properties of an object file before allowing it to execute. Java uses a byte-code verifier [40] to provide low-level security. Before loading a class, the verifier performs data-flow analysis on the class implementation to ensure that it is type safe, and that all control-flow instructions jump to valid locations. While Naccio/Win32's static analysis of executables is far less extensive, it allows the avoidance of easily-detectable dangerous behavior in untrusted code.

A more ambitious verification approach is Proof-Carrying Code (PCC) [21]. PCC combines a program with a proof that the program satisfies certain properties. Before installing the program, a certifier verifies the proof. In practice, PCC is limited by automatic proof-generation technology, and has been used most effectively only for simple properties [22]. The significant size of proofs and difficulty of generating proofs for more interesting policies limits the current usefulness of PCC.

**Software-Based Fault Isolation (SFI)** [37] [29] uses code transformation to limit all memory writes, reads (optionally), and control-flow instructions to specific segments of memory (potentially different for each type of instruction). This is done by inserting new instructions before each memory or control-flow instruction whose target address cannot be statically determined to be permissible. The additional instructions check at run-time that the target address is legal. SFI was originally designed to enable untrusted extension modules to be run in the memory space of trusted programs, such as an operating system kernel or web browser. Omniware [1] used SFI in a specialized compiler to provide low-level code safety in a mobile code system. MiSFIT [30] uses a specialized C++ compiler to provide SFI for the x86 processor platform. NAI labs is using a version of SFI applied at run-time to avoid bypassing of their Generic Software Wrapper system (described in Section 8.2) [8]. More details of SFI were presented in Section 5.1.1. A modified version of SFI, described in Section 5.1.2, is used to provide control-flow safety in Naccio/Win32.

## 8.2 High-Level Code Safety

Provided low-level code safety is in place, we can employ mechanisms for high-level code safety with confidence that they cannot be circumvented by forged pointers and jumps. A number of high-level code safety techniques have been invented, many directed toward mobile code [38] or extensible systems. Only those most relevant to this work are discussed here, including previous efforts based on wrappers, Java, Ariel, and systems based on Execution Monitoring and operating system extensions.

**Wrappers** applied to system calls are the implementation technique used by Naccio/Win32 to limit program access to resources, but Naccio/Win32 is not the first system to use wrappers for this purpose. Janus is a wrapper system intended to protect helper applications that may be subverted by data from an adversary [11]. It uses special debugging features of the Solaris operating system to cause wrapper code in a separate process to be invoked before and after selected system calls, but can only cause the original system call to continue unmodified or be denied, rather than being able to alter program behavior in a more general way such as replacing the system call with another. Other disadvantages of Janus are that it requires potentially expensive context switches for each wrapper, and is highly operating system specific.

Work at NAI Labs on Generic Software Wrappers (GSW) has used a loadable kernel module to implement wrappers for the Solaris and FreeBSD operating systems [9]. As kernel code, the GSW wrapper system requires no additional context-switches, and is fully protected from user code. GSW includes facilities to create wrappers that execute code before, after, or in place of of specified system calls, and even a small kernel-resident database for sharing data between wrappers. The kernel-based mechanism does have the disadvantage of imposing a small overhead (about 3%) on all programs running on the system, even those with no wrappers applied. A primary advantage of GSW is its Wrapper Description Language (WDL), which allows wrappers to be written in a simple way that provides some level of platform-independence. This is based on a platform description that specifies attributes of system calls and their return values and arguments (such as identifying paths, file descriptors, etc). This mechanism does not go nearly as

far as Naccio's abstract resources and platform interface, however, and thus is unlikely to provide the same degree of platform-independent capability. Nonetheless, the current C-based resource implementations and a Naccio platform interface written in WDL on each applicable platform could allow Naccio to be applied to additional platforms quickly using GSW rather than direct program transformation as the method of applying wrappers.

The current GSW platforms are very similar (both being UNIX variants), and it remains to be seen how well the current abstractions of WDL will extend to other platforms. A port of GSW to Windows NT is currently in progress, and applies wrappers to Win32 API functions as Naccio/Win32 does [31]. Since NT provides no equivalent of the loadable kernel module, the wrappers are applied by modifications to address tables at run-time. Currently, though, no low-level protection exists to make these wrappers non-bypassable. Work is in progress to develop a system based on SFI to provide this, focusing on application of SFI transformations at run-time to avoid the difficulties of disassembly of x86 binaries as well as potential licensing issues with the transformation of programs [8]. When this work is complete, it could provide a suitable implementation of most of the protective transformations needed by Naccio/Win32.

**Java** and its run-time environment provide high-level security by allowing access to system resources only through the Java API. The library functions are implemented so that code safety checks are performed before certain system calls are executed [10]. Security policies are defined by creating custom Java classes that implement those checks [6]. JDK 1.2 has introduced more flexible means for applying different levels of security to different portions of code [12]. All of Java's mechanisms, however, are limited by the points at which API functions make calls to security checking, leaving a user with no way of enforcing policies on functions that call no checks. For instance, while Java provides a hook for security checking when a file is opened (allowing arbitrary security policies on which files may be opened), it provides no such opportunity when individual bytes are written to files, making it impossible to enforce security policies that limit the number or contents of bytes written. The primary reason this limitation remains is the fixed overhead associated with a Java security call, even if no policy is being enforced. Naccio's resource descriptions and platform interface provide enough flexibility to allow checking to be performed on any

set of system calls, without the policy author needing to be concerned with the details of precisely which system calls are being affected. Instead, the policy author reasons abstractly about which resources should be protected, and the policy compiler inserts safety checking at appropriate points, introducing no overhead at points where no checking is required for a particular policy.

**The Ariel Project** describes policies using a declarative language and enforces them by inserting code into Java classes. The transformations performed by Ariel [23] to enforce a policy are similar to those used by Naccio/JavaVM, the Naccio system for Java bytecodes. Because of the declarative nature of its policy descriptions, Ariel is unable to describe policies that react to violations by modifying program behavior, as can be expressed using Naccio's mechanisms. Policies are described at the level of the Java API so they are not portable across platforms, and creating a policy requires intimate knowledge of the API rather than utilizing a more abstract notion of resources.

**Execution Monitoring** is the method of enforcement utilized by the class of enforcement mechanisms that Schneider dubs EM [28]. They enforce security policies by monitoring a target system and terminating an execution immediately before the policy would be violated. Enforcement mechanisms in class EM can only enforce security policies that are safety policies (those that can be defined as a predicate on a prefix of execution states). Naccio is not in class EM because it modifies the application instead of simply monitoring an execution. However, with restrictions on the platform interface, safety property definitions, and static analyses performed by the application transformer, Naccio can be viewed as an execution monitor in class EM. Naccio policies that do not satisfy these restrictions can change the behavior of the program in more fundamental ways and are harder to classify, but also potentially more useful.

Schneider suggests techniques for using finite-state automata to express safety policies enforceable by execution monitoring. Úlfar Erlingsson has developed a Security Automata SFI Implementation (SASI) [36], a system that enforces policies defined using automata by inserting code in program executables (x86 assembly). Although no performance analysis is available, SASI should be able to enforce many policies more efficiently than Naccio since it does not require the overhead associated with maintaining run-time objects corresponding



to abstract resources. The main advantage of Naccio over SASI is that it offers a convenient, platform-independent method of defining policies.

**Operating Systems** provide security by starting from resources and determining the privilege level of each program that attempts to manipulate those resources, rather than starting from the program and trying to constrain its effect on system resources as Naccio's transformations do. Naccio's high-level methodologies could equally well be applied to an operating system centered approach. The application centered approach was chosen because it eliminates ties to a particular operating system, and provides a more general platform for exploring research issues. It is also easier to deploy a code transformer than an operating system kernel modification.

At a coarse level, modern operating systems already constrain the use of resources. For example, UNIX prevents programs from violating file system permissions. A few projects have attempted to extend this in a way that restricts what specific programs or processes may do. Wichers et al. suggest protecting a system from malicious programs using per-file access control lists [39]. Another project, TRON [2], is a process-specific file protection system for the UNIX operating system. It allows users to create shells with specific access permissions that apply to all programs executed in the shell. The disadvantage of operating-system based solutions is that they are likely to have an impact on trusted applications as well as on untrusted ones. Naccio's transformations include all safety checking within the application itself, thus having no effect on applications that are not running under a safety policy.

### **8.3 Binary Editing**

Most work in binary code editing has been directed toward the purposes of instrumentation or optimization. Instrumentation of executables inserts instructions that allow data to be gathered about the executable's run-time behavior, such as memory accesses, procedure calls, etc. Binary editing has also been used for optimization of executables, through removal of redundant code, code reordering to enhance cache performance, interprocedural optimization of data flow, and other methods.

Key challenges in binary editing include code discovery, relocations, and handling of indirect jumps. Code discovery is the process by which code is distinguished from data. Since code and data are often interspersed in memory, it is important that code be identified for modification, while data is left unchanged, lest the modified program behave incorrectly. Relocations are required to fix up addresses used by jump instructions and memory accesses, based on the fact that the location of an instruction or piece of data in the modified program may not be identical to its location in the original program. Indirect jump instructions (those which use the contents of a register or memory location as their target) compound the problem because their target often cannot be determined at transform time. Most binary editors make use of some level of added information in the executable beyond raw code and data, ranging from relocation information (often included in executables for the purpose of the loader), to a full-fledged debugging table that identifies procedure entry points and data regions. If any of these tools are to be used for security-sensitive purposes such as Naccio, the use of such additional information in code modification must be minimized and carefully checked, so that inaccuracies in that information cannot introduce security flaws. In general, this must be done by some sort of verification of any additional information used, or by making worst-case assumptions in cases where properties of the program cannot be determined exactly.

While most binary editing tools developed have been specialized tools for a single purpose, a number of attempts have been made to develop more general binary editing systems. EEL [17] took this goal farthest, with a platform-independent binary editing system. EEL translates fully-linked programs into a platform-independent register transfer language (RTL), and allows modifications to be performed on RTL before translating the program back to machine instructions. In theory, the same transformation rules can be applied to any machine platform through the use of RTL, but EEL has as yet been implemented only for SPARC executables running under SunOS and Solaris, and thus its promise of portability is unrealized (though a partial port for RS6000 AIX exists). EEL's RTL is also very RISC-like, and thus it remains to be seen whether EEL could successfully support other processor types such as x86.

DYNINST is a system for run-time instrumentation that has been successfully deployed

on the x86, Alpha, PowerPC, and SPARC platforms [13] [14]. It adds instrumentation at run-time by replacing instructions with fixed jumps to instrumentation routines, in a way similar to the application of SFI checking in Naccio/Win32. DYNINST uses a platform-independent language called MDL that is tailored to program instrumentation and allows instructions and data of the types needed by instrumentation to be handled easily.

Like EEL, OM [33] also makes use of an RTL in performing link-time modifications on the Alpha platform. Relocation information in object files is used to aid in the modifications. ATOM [34] is a framework built to simplify the process of program instrumentation using OM. ATOM provides a simple set of APIs for instrumenting programs and navigating the structure of a program, as well as being distributed with several standard instrumentation tools. Unlike OM, ATOM does not allow arbitrary modifications to an executable; in particular it does not allow instructions from the original executable to be deleted or re-ordered. ATOM parses fully compiled executables, but requires that relocation information be included by the compiler.

Like OM, ATOM supports only Alpha executables, but has been successfully deployed for the OSF/1, Digital UNIX, and Windows NT operating systems. The NT version of ATOM was based on libraries developed for Spike[4], an instrumentation and optimization tool for NT executables. Along with its binary editing, Spike also intercepts system calls via replacement of DLLs to transparently substitute instrumented executables and DLLs for their unmodified versions. While the number of calls wrapped by Spike is small, its methods of doing so are similar to those used in Naccio/Win32. Spike and its component libraries would be a suitable tool for the instruction-level transformations described in Chapter 5, required by Naccio to ensure that higher level safety mechanisms are not bypassed.

Another Win32 binary editing tool is Etch [27], a general tool for instrumentation and optimization of Win32 executables on the x86 platform. Microsoft is also currently researching similar technology under the name Vulcan [32]. Neither Etch nor Vulcan is currently available for research purposes.

# Chapter 9

## Conclusions

While the design of Naccio/Win32 is complete, and the prototype implementation proves key features of the design, there is still much work to be done. Section 9.1 surveys some of the tasks that remain and suggests directions that may be taken by future research. Section 9.2 concludes by summarizing and evaluating major accomplishments of Naccio/Win32.

### 9.1 Future Work

Many significant research and implementation tasks remain for future work on Naccio/Win32 and Naccio in general. The most obvious task for Naccio/Win32 is to implement aspects of the design not included in the current prototype. Full implementation and testing of the protective transformations would provide a much clearer picture of the potential of Naccio/Win32 to protect against malicious code produced by knowledgeable adversaries. By taking advantage of the existing mechanisms for application of wrappers, many of the protections required of the protective transformations can be achieved in a simple and processor-independent way as described in Chapter 5. The use of SFI for control-flow protection must, however, be processor-specific, and different challenges will have to be met for each processor architecture attempted.

The largest open issue in the Naccio/Win32 design is multithreaded memory protection. As described in Section 5.2, the memory safety portion of the protective transformations can, in its current form, guarantee protection only for applications with a single thread

of control. Section 5.2.2 described possible approaches to providing proper protective transformations for multithreaded applications, but more investigation and testing is needed to determine which approach will produce the best results.

The needs of the protective transformations will also help to drive work in the analysis and optimization of wrapper source code derived from the platform interface. The current implementation includes no such optimizations, and only relatively simple optimizations of resource implementations. Thus there is much more potential for work in improving the policy-specific optimizations performed by the policy compiler, as well as efficient ways of combining the standard functionality needed by protective transformations with the custom functionality described by the platform interface.

The other main limitation of the current Naccio/Win32 prototype is its support for only a limited subset of Win32 API functionality. A more complete platform interface would allow Naccio/Win32 to protect a broader range of resources used by a wider variety of applications. Supplemental APIs and programming methodologies such as MFC, DirectX, and COM are also deserving of individual consideration. Such supplemental APIs could be supported indirectly by treating their implementations as untrusted code, but there could be benefits to the performance, flexibility, clarity, and accuracy of policy enforcement if they are included in the platform interface and supported directly. Tools that could automatically generate platform interfaces for supplemental APIs based on core APIs and source code are worthy of future investigation.

No security system can be trusted until it has been tested and used extensively. Automated verification systems to ensure the safety of policies or platform interfaces would be a highly useful tool for Naccio, and worthy of investigation. Since the danger of vulnerabilities introduced by bugs or flaws in design or implementation is ever-present, however, it is important for a security system to receive public scrutiny and testing before it is trusted for the protection of vital systems. The Naccio/JavaVM prototype is currently undergoing such scrutiny through a World Wide Web site that allows remote users to upload code and attempt to attack the system by bypassing Naccio safety policies on file system and network use. Prizes are offered to any who can successfully bypass security as an incentive. Once its low-level code safety and support for the Win32 API is more complete, Naccio/Win32 could

be tested in a similar fashion. A more extensive test would require large-scale deployment of Naccio/Win32 in a wider variety of situations.

A large part of the promised benefit of Naccio is the use of simple, platform-independent safety policies. Win32 is the second platform for which Naccio was implemented, and thus Naccio/Win32 is the first test of this promise. For that promise to be fully realized, further platforms should be implemented. It is estimated that the work involved in creating the policy compiler and application transformer for each platform will be significantly less once a small number of platforms have been implemented. Most modern operating systems share many standard design principles so the enforcement of policies on any operating system will share many of the same challenges both in the application of policy transformations, and in the type of protective transformations that must be performed. Thus much design work from Naccio/Win32 should be applicable to other operating system platforms. Still, care will need to be taken to ensure that the subtleties of an individual platform are properly accounted for, and the task of creating a complete platform interface will be significant and unique for each new set of system calls.

## **9.2 Summary and Evaluation**

This thesis has described Naccio/Win32, which applies the Naccio architecture to the enforcement of safety policies on Win32 executables. Naccio provides a platform-independent architecture for specifying safety policies as limitations on the use of abstract resources, and for the enforcement of those safety policies by transforming programs. Naccio/Win32 enforces policies by constraining the use of the Win32 API, and uses transformations to provide the low-level code safety required to ensure that those constraints cannot be bypassed. A prototype of Naccio/Win32 has been implemented and tested, and results described here have shown that this enforcement can be achieved without unreasonable costs in time or effort.

The Naccio architecture and Naccio/Win32 were designed to provide robust code safety with the three key goals of flexibility, usability, and efficiency as primary concerns. Naccio's simple, platform-independent policy-description mechanisms allow a broad range of safety

policies to be expressed easily, including policies tailored for a particular application or environment, or introduced to respond to a newly discovered threat. Platform-specific application transformers like Naccio/Win32 allow platform-independent policies to be enforced on applications on many different platforms, while encapsulating the platform-specific knowledge needed to ensure that policies cannot be bypassed by hostile code.

Using transformation to integrate safety checking into applications themselves provides many benefits. Since safety checking is confined to the application, trusted applications experience no overhead. Transformations can be applied to fully-compiled programs in native executable format, eliminating all of the problems inherent in re-compilation, or the use of an interpreted language. Pre-compilation of policies to native code allows transformation times to be small, minimizing the noticeable time before a program can be run, and policy-specific optimizations allow the safety checking to be limited to exactly the requirements of the policy to be enforced, eliminating unnecessary overhead at run-time.

The primary means of enforcement used by Naccio/Win32 is attaching checking to the Win32 API of system functions. This API is well-documented, allowing it to be thoroughly described in the platform interface, and also well separated from application code by the interface between an application and its DLLs, making the application of wrappers easy. Because of that separation, the transformations applied to application code are limited to import table modifications, and low level protective transformations that ensure that wrapper-based protections cannot be bypassed. These protective transformations themselves use the wrapper-based protections to good effect, and the remainder of their work is performed by instruction modifications based on Software-Based Fault Isolation [37].

Naccio/Win32's design achieves its three key goals in all of these ways, and the current Naccio/Win32 prototype provides a proof of concept that shows that they can be achieved for real applications and real safety needs. The potential usefulness and effectiveness of Naccio/Win32 is clear in its design, and the methodologies introduced by Naccio and Naccio/Win32 can provide great benefits to users in the future. It is hoped that through further research this potential can be further explored and realized, and that the ideas presented here will aid in protecting users from the dangers of the modern computing environment, allowing them to take better advantage of it in the future.

If Naccio/Win32 or a similar system were put into large-scale use, users would be able to download and execute code freely without any worry of damage caused by malicious or buggy programs. Transformations would be applied automatically as part of a browser or program installer, with policies selected by a system administrator or automated tool, or by the user in order to provide safety custom-tailored to the needs of the user and the application. The safe distribution of code will increase the safety of current uses of mobile code (such as web-based applets), and encourage further uses such as component-based software downloaded on-demand, active content through programs included in documents, or automatically retrieved program updates and content viewers. Code safety should ideally be applied even to trusted non-mobile applications, in order to protect users from bugs and increase their confidence in the safety of their data.

The uses of code safety are many and varied if it can be flexible, efficient, and easy to use enough to meet every demand that arises. The Naccio architecture meets those goals, and by putting Naccio to use on the dominant system platform Naccio/Win32 demonstrates that the promise of flexible code safety is not an empty one, but an opportunity to make the computing environments of tomorrow safe for everybody.



# Appendix A

## Sample Safety Policies

This appendix includes the source code of the sample policies described in Section 7.1, and used in testing of the Naccio/Win32 prototype. The full resource descriptions are included first to aid in understanding of the policies.

### A.1 Resource Descriptions

The `RFileSystem` and `RFile` resource descriptions are as described in Section 2.2.1, except for the inclusion of the previously-elided operations.

#### The `RFileSystem` Resource Description

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

  openRead (file: RFile)
    "Called before a file is opened for reading."
  openCreate (file: RFile)
    "Called before a new file is created for writing."
  openWrite (file: RFile)
    "Called before an existing file is opened for writing."
  openAppend (file: RFile)
    "Called before an existing file is opened for appending."
  close (file: RFile)
    "Called before a file is closed."

  write (file: RFile, n: int)
    "Called before n bytes are written to a file."
  preRead (file: RFile, n: int)
    "Called before up to n bytes are read from a file."
  postRead (file: RFile, n: int)
    "Called after n bytes were read from a file."
```

```

delete (file: RFile)
    "Called before a file is deleted."
makeDirectory (file: RFile)
    "Called before creating a directory."
rename (file: RFile, newfile: RFile)
    "Called before renaming the specified file."
copy (file: RFile, newfile: RFile)
    "Called before copying the specified file."

observeExists (file: RFile)
    "Called before revealing if a named file exists."

observeExists (file: RFile)
    "Called before revealing if a file exists."
observeIsFile (file: RFile)
    "Called before distinguishing a file from a directory."
observeLength (file: RFile)
    "Called before revealing the length of a file."
observeList (file: RFile)
    "Called before revealing the list of files in a directory."

observeLastModifiedTime (file: RFile)
    "Called before revealing when a file was last modified."
setLastModifiedTime (file: RFile)
    "Called before setting the last modified time of a file."
observeLastAccessTime (file: RFile)
    "Called before revealing when a file was last accessed."
setLastAccessTime (file: RFile)
    "Called before setting the last accessed time of a file."
observeCreationTime (file: RFile)
    "Called before revealing when a file was created."
setCreationTime (file: RFile)
    "Called before setting the last creation time of a file."
observeAttributes (file: RFile)
    "Called before revealing file attributes."
setAttributes (file: RFile)
    "Called before setting file attributes"

```

## The RFile Resource Description

```

resource RFile
  operations
    RFile (pathname: String)
        "Called to construct a new RFile object."

    finalize ()
        "Called when finished with an RFile object."

```

## A.2 Sample Policies

As described in section 2.3, a safety policy is made up of a combination of safety properties with parameters. The sample safety properties used in testing the Naccio/Win32 prototype are listed here, along with the state blocks that they use to maintain state information.

The `Null` policy includes no safety checking code at all. For all but the `Combined` policy, the other policies used included only a single property, so their policy code is not included here. The code for the `Combined` policy is a simple combination of properties as shown below.

```
policy Combined {
  NoOverwrite
  LimitBytesWritten(100000000)
  LimitFilePath("d:\\legal")
  SetReadOnlyDir("d:\\legal\\readonly")
}
```

### A.2.1 Safety Properties

Note that the `matchesPathPrefix`, `getFileSize`, and `fileExists` functions are provided by the Naccio library.

#### The `LimitPath` Safety Property

```
property LimitPath (path: String) {
  requires FileNames;

  precheck RFile.RFile (pathname: String) {
    if (!matchesPathPrefix(pathname,path)) {
      violation("Attempt to access illegal file " + pathname +
        ". Only files in the subtree " + path +
        " may be accessed.");
    }
  }
}
```

## The ReadOnlyDir Safety Property

```
property ReadOnlyDir (path: String) {
  requires FileNames;

  precheck RFileSystem.openWrite(file: RFile),
    RFileSystem.openCreate(file: RFile),
    RFileSystem.openAppend(file: RFile),
    RFileSystem.preDelete(file: RFile),
    RFileSystem.setLastModifiedTime(file: RFile),
    RFileSystem.setLastAccessTime(file: RFile),
    RFileSystem.setCreationTime(file: RFile),
    RFileSystem.setAttributes(file: RFile),
    RFileSystem.makeDirectory(file: RFile) {
    if(matchesPathPrefix(file.name,path)) {
      violation("Attempt to write file " + file.name +
        " in the read-only subtree " + path + ".");
    }
  }

  precheck RFileSystem.rename(file: RFile, newfile: RFile) {
    if(matchesPathPrefix(file.name,path)) {
      violation("Attempt to write file " + file.name +
        " in the read-only subtree " + path + ".");
    }
    if(matchesPathPrefix(newfile.name,path)) {
      violation("Attempt to write file " + newfile.name +
        " in the read-only subtree " + path + ".");
    }
  }

  precheck RFileSystem.copy(file: RFile, newfile: RFile) {
    if(matchesPathPrefix(newfile.name,path)) {
      violation("Attempt to write file " + newfile.name +
        " in the read-only subtree " + path + ".");
    }
  }
}
```

The alternate version of the `ReadOnlyDir` policy used for testing with `pkzip` moved checking from open operations into the write operation simply by changing the declaration of the first `precheck` clause. The `openWrite` and `openAppend` operations are removed and the `write` operation is added. The code of the check remains the same. The `openCreate` operation is still checked, since creating a file should count as writing even if no bytes are written.

## The NoOverwrite Safety Property

The `NoOverwrite` property differs from its description in Section 2.2.1 only in the inclusion of previously elided resource operations.

```

property NoOverwrite {
    requires FileNames;

    precheck RFileSystem.openWrite(file: RFile),
        RFileSystem.openAppend(file: RFile),
        RFileSystem.preDelete(file: RFile),
        RFileSystem.setLastModifiedTime(file: RFile),
        RFileSystem.setLastAccessTime(file: RFile),
        RFileSystem.setCreationTime(file: RFile),
        RFileSystem.setAttributes(file: RFile),
        RFileSystem.rename(file: RFile, newfile: RFile) {
        violation("Attempt to affect existing file " + file.name + ".");
    }
}

precheck RFileSystem.copy(file: RFile, newfile: RFile) {
    if(fileExists(newfile.name)) {
        violation("Attempt to affect existing file " + file.name + ".");
    }
}
}

```

The alternate version of the `NoOverwrite` policy used for testing with `pkzip` moved checking from open operations into the write operation, using the state variable from the `CreationMark` state block to keep track of which files were pre-existing. The `requires` clause and the declaration of the first `precheck` clause are changed, and the new check is added to the write operation.

```

property NoOverwrite {
    requires FileNames, CreationMark;

    precheck RFileSystem.preDelete(file: RFile),
        RFileSystem.setLastModifiedTime(file: RFile),
        RFileSystem.setLastAccessTime(file: RFile),
        RFileSystem.setCreationTime(file: RFile),
        RFileSystem.setAttributes(file: RFile),
        RFileSystem.rename(file: RFile, newfile: RFile) {
        violation("Attempt to affect existing file " + file.name + ".");
    }
}

precheck RFileSystem.copy(file: RFile, newfile: RFile) {
    if(fileExists(newfile.name)) {
        violation("Attempt to affect existing file " + file.name + ".");
    }
}

precheck RFileSystem.write(file: RFile, nbytes: int) {
    if(!file.created) {
        violation("Attempt to affect existing file " + file.name + ".");
    }
}
}

```

## The LimitBytesWritten Safety Property

```
property LimitBytesWritten (n: int) {
  requires TrackTotalBytesWritten, FileNames;

  precheck RFileSystem.write (file: RFile, nbytes: int) {
    if (bytes_written > n) {
      violation("Attempt to write more than " + n + " bytes. Writing " +
        nbytes + " to " + file.name + ".");
    }
  }
  // Copying is treated as writing the full size of the file.
  precheck RFileSystem.copy (file: RFile, newfile: RFile) {
    if (bytes_written > n) {
      violation("Attempt to write more than " + n + " bytes writing " +
        getFileSize(file.name) + " bytes copying " + file.name +
        " to " + newfile.name + ".");
    }
  }
}
```

## A.2.2 State Blocks

### The FileNames State Block

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

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

### The TrackTotalBytesWritten State Block

The TrackTotalBytesWritten state block differs from its description in Section 2.2.1 only in the addition of handling for the copy operation.

```
stateblock TrackTotalBytesWritten augments RFileSystem {
  requires FileNames;
  addfield bytes_written: int = 0;

  precode write (file: RFile, nbytes: int) {
    bytes_written += nbytes;
  }

  precode copy (file: RFile, newfile: RFile) {
    bytes_written += getFileSize(file.name);
  }
}
```

## The CreationMark State Blocks

These stateblocks keep track of created files for the alternate version of the `NoOverwrite` property.

```
stateblock CreationMark augments RFileSystem {
  requires RFileCreationMark;

  precode openCreate(file: RFile) {
    file.created = true;
  }

  precode makeDirectory(file: RFile) {
    file.created = true;
  }
}

stateblock RFileCreationMark augments RFile {
  addfield created: boolean;

  precode RFile(pathname: String) {
    created = false;
  }
}
```

# Bibliography

- [1] Aali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and Language-Independent Mobile Programs. In *ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, May 1996.
- [2] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Winter USENIX*, 1995.
- [3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *15th ACM Symposium on Operating System Principles*, 1995.
- [4] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An Optimizer for Alpha/NT Executables. In *USENIX Windows NT Workshop*, August 1997.
- [5] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.
- [6] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *IEEE Symposium on Security and Privacy*, May 1996.
- [7] David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*, May 1999.
- [8] Mark S. Feldman. Using Software Fault Isolation to Enforce Non-Bypassability. Presentation at IEEE Symposium on Security and Privacy, May 1999.



- [9] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *IEEE Symposium on Security and Privacy*, May 1999.
- [10] J. Steven Fritzinger and Marianne Mueller. Java™ Security. Technical report, Sun Microsystems, Inc., 1996.
- [11] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *6th USENIX Security Symposium*, 1996.
- [12] Li Gong, Marianne Mueller, and Hemma Prafullchandra. Going Beyond the Sandbox: An Overview of the new Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [13] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Scalable High Performance Computing Conference*, May 1994.
- [14] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Gongalves, Oscar Niam, Zhichen Xu, and Ling Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *PACT'97*, November 1997.
- [15] Dexter Kozen. Efficient Code Certification. Technical Report 98-1661, Cornell University, January 1998.
- [16] Butler Lampson. Protection. In *Fifth Princeton Symposium on Information Sciences and Systems*, March 1971.
- [17] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1995.
- [18] Barbara Liskov, R. Atkinson, T Boom, E Moss, J Schaffert, R Scheifler, and A Snyder. *CLU Reference Manual*. Springer-Verlag, 1981. Number 114 in Lecture Notes in Computer Science.

- [19] Microsoft Corporation. *MSDN Library*, January 1999.
- [20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. (originally published in 1990).
- [21] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating Systems Design and Implementation*, 1996.
- [22] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 1998.
- [23] Raju Pandey and Bryant Hashii. Providing Fine-Grained Access Control for Mobile Programs Through Binary Editing. Technical Report TR98-08, UC Davis, August 1998.
- [24] Matt Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. *Microsoft Systems Journal*, 9(3):15–34, March 1994.
- [25] Matt Pietrek. Windows Q&A. *Microsoft Systems Journal*, 10(7):83–91, July 1995.
- [26] Matt Pietrek. Windows Q&A. *Microsoft Systems Journal*, 10(8):91–96, August 1995.
- [27] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *USENIX Windows NT Workshop*, August 1997.
- [28] Fred B. Schneider. Enforceable Security Policies. Technical Report TR98-1664, Cornell University Computer Science, January 1998.
- [29] Scott M. Silver. Implementation and Analysis of Software Based Fault Isolation. Technical Report PCS-TR96-287, Dartmouth College, June 1996.
- [30] Christopher Small and Margo Seltzer. MiSFIT: A Tool for Constructing Safe Extensible C++ Systems. In *Third Conference on Object-Oriented Technologies and Systems*, 1997.

- [31] Larry Spector and Lee Badger. Porting Wrappers from UNIX to Windows NT: Lessons Learned. Presentation at IEEE Symposium on Security and Privacy, May 1999.
- [32] Amitabh Srivastava. Indirect personal communication, September 1998.
- [33] Amitabh Srivastava and Alan Eustace. A Practical System for Intermodule Code Optimization at Link-Time. Technical Report 92/6, Digital Western Research Laboratory, December 1992.
- [34] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. Technical Report 94/2, Digital Western Research Laboratory, March 1994.
- [35] Sun Microsystems. *The Java Language: An Overview*, 1996.
- [36] Úlfar Erlingsson. Personal communication, March and May 1999.
- [37] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Symposium on Operating System Principles*, 1993.
- [38] David Wetherall. Safety Mechanisms for Mobile Code. Area Examination Paper, Telemedia Networks and Systems Group, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1995.
- [39] D. R. Wichers, D. M. Cook, R. A. Olsson, J. Corssley, P. Kerchen, K. Levit, and R. Lo. An Access Control List Approach to Anti-viral Security. In *13th National Computer Security Conference*, October 1990.
- [40] Frank Yellin. Low-level Security in Java. In *WWW4 Conference*, 1995.