To appear in *IEEE Security and Privacy*, Oakland, CA. May 9-12, 1999.

Flexible Policy-Directed Code Safety

David Evans Andrew Twyman
evs@sds.lcs.mit.edu twyman@sds.lcs.mit.edu
MIT Laboratory for Computer Science

Abstract

This work introduces a new approach to code safety. We present Naccio, a system architecture that allows a large class of safety policies to be expressed in a general and platform-independent way. Policies are defined in terms of abstract resource manipulations. We describe mechanisms that can be used to efficiently and conveniently enforce these safety policies by transforming programs. We are developing implementations of Naccio that enforce policies on JavaVM classes and Win32 executables. We report on results using the JavaVM prototype.

1 Introduction

> The security system was adequate, but it did not foresee an armed robbery.
>
> Italian Minister of Culture Walter Veltroni, explaining the theft of two van Goghs and a Cézanne from Rome's National Gallery.

Code safety means knowing what a program will not do. The problem of code safety has been around since the earliest days of computing, but has become increasingly important as it has become easier to distribute programs. Current environments demand more flexible security than traditional operating systems provide. Users should be able to run different programs with varying degrees of trust and specific restrictions and capabilities.

Most code safety systems work adequately until they are attacked in ways the system designers did not foresee. What is needed is a system that allows new safety policies that enforce constraints outside those considered by the original system designers to be rapidly created and deployed in response to new threats.

This paper introduces Naccio, a platform-independent architecture for code safety designed to provide superior flexibility. While no security system can foresee all possible attacks, by providing a system that can be used to define and enforce a wide range of policies we hope to be able to quickly respond to new threats. Naccio can define and enforce policies that place arbitrary constraints on resource manipulations as well as policies that alter how a program manipulates resources, but cannot define or enforce liveness properties or policies that depend on structural properties of the code.

The next section presents the Naccio architecture. Conceptually, Naccio takes a program and a safety policy and produces a program that behaves similarly to the original program except that it is guaranteed to satisfy the safety policy. Section 3 explains how resources are described and Section 4 shows how we use those resource descriptions to define safety policies. In Section 5, we show how a specific platform is specified in terms of how it manipulates resources. Section 6 discusses issues involved in developing Naccio implementations for the JavaVM and Win32 platforms. Section 7 reports on results using a prototype implementation to enforce safety policies on JavaVM applications. In Section 8, we survey related work. The final section offers some conclusions.

2 System architecture

Anecdotal evidence suggests that any code safety system that places a burden on its users will be quickly disabled, since its benefits are only apparent in the extraordinary cases in which a program is behaving dangerously. Most users will not create new safety policies, but will select from a list of predefined policies or use default settings chosen by a system administrator. Hence, a primary design goal is that predefined safety policies can be enforced cheaply and effortlessly.

Safety policies will be written mostly by experts and distributed both with the system and in response to new threats. It is important, however, that system administrators and sophisticated users can create and modify policies to respond to specific needs or threats. Safety policy authors can be expected to spend some time learning to read and write policies, but should not be required to understand details of one or more target platforms. Hence, it is important that we allow safety policies to be described in a manner that hides the complexities and details of a particular platform.

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

Instead, we could write wrapper functions that perform the necessary checks and then call the original system functions. To enforce the policy, we would modify the target program to call the wrapper functions instead of the protected system calls. Though wrappers are a reasonable implementation technique, they are not appropriate for describing safety policies since creating or understanding them requires intimate knowledge of the underlying system. To implement the write limit policy, the author of the safety policy would need to identify and understand every system call that may write to a file. For even a supposedly simple platform like the Java API, this involves knowing about dozens of different methods. Changing the policy would require editing the wrappers, and there would be no way to use the same policy on other platforms.

Our solution is to express safety policies at a more abstract level and to provide a tool that generates the wrappers needed to enforce a policy on a particular platform. We express safety policies in terms of abstract resource manipulations and characterize a platform by how its system calls affect those resources.

Figure 1 shows the Naccio system architecture. It is divided into a *policy generator* and an *application transformer*. A policy author runs the policy generator to produce what the application transformer uses to enforce the policy on a particular program. Since policy generation is a relatively infrequent task, we trade off execution time of the policy generator to make application transformation fast and to reduce the run-time overhead associated with safety checks. Once a policy has been generated, it can be reused for each application on which we want to enforce the policy.

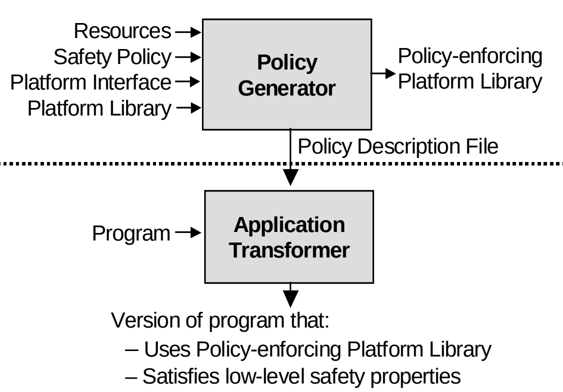

Figure 1. Naccio Architecture. The top half of the figure depicts what a policy author does to generate a new policy. The bottom half shows what happens the first time a user elects to execute a given program enforcing that policy.

The inputs to the policy generator are:

- *Resource descriptions* (Section 3) – abstract descriptions of system resources.
- The *safety policy* (Section 4) – a description of the constraints to be enforced on resource manipulations.
- The *platform interface* (Section 5) – a description of a particular platform that describes how its system calls manipulate resources.
- The *platform library* – the unaltered platform library (e.g., Java API classes or Win32 system DLLs).

The policy generator produces a *policy-enforcing platform library*, a version of the platform library that includes checking code necessary to enforce the policy. It also produces a *policy description file* that contains transformation rules required to enforce the policy.

The application transformer is run when a user elects to enforce a particular policy on an application. It reads a policy description file and a target program and performs the directed transformations to produce a version of the program that is guaranteed to satisfy the safety policy. This involves replacing system calls in the program with calls to the policy-enforcing library. For each program and selected policy, we need to run the application transformer only once. Afterwards, the resulting program can be executed normally.

In addition, the application transformer must ensure that the resulting program satisfies the low-level code safety properties necessary to prevent malicious programs from circumventing the high-level code safety mechanisms. At a minimum, it must prevent programs from modifying their own code, writing to storage used in safety checking, or jumping to arbitrary memory locations that could contain system library code. Although our work relies on low-level code safety to ensure the integrity of high-level code safety mechanisms, our focus is on providing better ways to define policies that constrain the use of system resources. Various techniques for low-level code safety (such as bytecode verification or software fault isolation, see Section 8.1) can be used to provide the necessary low-level code safety properties.

A Naccio implementation is characterized by the format of the input program and format and content of the platform libraries it uses. We are developing Naccio implementations that enforce safety policies on JavaVM classes and Win32 executables.

3 Describing resources

Resource descriptions provide a way to identify resources and the ways they are manipulated. Examples of resources include files, network connections, threads and displays. Resource descriptions are platform-independent, but they may describe platform-specific resources such as the Windows registry. Policy authors read resource descriptions, but typically do not need to modify them.

We describe resources by listing their operations. Resource descriptions have no state or implementation. They are merely hooks for use in defining safety policies. The meaning of a resource operation is indicated by informal documentation. The essential promise is that a transformed program will invoke the related resource operation with the correct arguments whenever a particular event occurs. It is up to the policy generator and platform interface to ensure that this is the case.

Figure 2 shows resource descriptions for the file system. The global modifier indicates that only one RFileSystem instance exists for an 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 identifies a particular file.

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

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

  openRead (file: RFile)
    Called before 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 existing file is opened for appending.
  close (file: RFile)     Called before file is closed.

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

  delete (file: RFile)   Called before file is deleted.

  observeExists (file: RFile)
     Called before revealing if file exists.
  observeWriteable (file: RFile)
    Called before revealing if file is writeable.

  … // other similar observe<X> operations elided

resource RFile
  RFile (pathname: String)
     Constructs object corresponding to pathname
```

Figure 2. File System Resource.

```
policy LimitWrite
  NoOverwrite, LimitBytesWritten (1000000)

property NoOverwrite
  check RFileSystem.openWrite (file: RFile),
       RFileSystem.openAppend (file: RFile),
       RFileSystem.delete (file: RFile)
     violation ("Attempt to overwrite file.");

property LimitBytesWritten (limit: int)
  requires TrackTotalBytesWritten;
  check RFileSystem.write (file: RFile, n: int)
     if (bytes_written > limit) violation ("Attempt to write …");

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

Figure 3. LimitWrite Safety Policy.

4 Defining safety policies

Safety policies are defined by attaching checking code to resource operations. A policy consists of any number of safety properties that place constraints on resource manipulations. Policies are described in a platform-independent way, but may be designed for platform-specific threats (e.g., a Unix-specific policy may restrict reading /etc/passwd).

Figure 3 shows the LimitWrite safety policy that instantiates two safety properties. NoOverwrite disallows replacing or altering the contents of any existing file. LimitBytesWritten (1000000) places a limit of one million on the total number of bytes that may be written to the file system. LimitWrite would not be a wise policy to use on an untrusted application since it does not constrain what files the application may read or how the application may use other resources such as the network.

A safety property consists of check clauses that attach checking code to resource operations. The check clause of the NoOverwrite property identifies the two RFileSystem resource operations called before an existing file is opened for writing (openWrite and openAppend) and the operation associated with deleting a file. The checking code 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. Although the checking code is written in a Java-like language, it is platform-independent and the same policy can be used on multiple platforms. The policy generator translates the checking code into the appropriate code for a particular platform.

The LimitBytesWritten property illustrates how a more complex safety property is defined. To enforce a limit on the number of bytes that may be written, the property must keep track of the total number of bytes written. This is done by the TrackBytesWritten state block that is referred to by the requires clause. TrackBytesWritten adds a field to the RFileSystem resource, and defines a precode action for the write operation. The body of the precode action will happen before all checking code associated with the resource operation. Hence, the value of bytes_written used in the LimitBytesWritten property is the number of bytes that will have been written if the upcoming write is allowed to execute. We keep the state maintenance and property checking code separate, since many safety properties use the same state.

The range of safety policies that can be defined is limited by the resource operations. Naccio can detect violations and observe and modify state only at execution points corresponding to resource operations. Using the RFileSystem resource shown in Figure 2, we could not detect a violation after a file write has occurred since the only resource operation associated with writing is called before the write occurs.

We are also limited by what information is passed to resource operations. Since RFileSystem.write takes an integer parameter revealing the number of bytes to be written but does not have a parameter corresponding to the actual data written, we cannot write a policy that constrains the actual values of bytes that may be written.

5 Describing platforms

In order to enforce a safety policy, the appropriate resource operations must be called as documented in the resource descriptions. The platform interface describes how system calls manipulate resources. Platform interfaces are tied to a particular platform and set of resource descriptions. For typical policies, policy authors should not need to look at or alter platform interfaces. Some modifications to platform interfaces may be necessary to define policies that alter program behavior in more substantial ways than simply detecting violations.

For a given execution platform, there may be several possible levels at which the platform interface could be defined. The level of the platform interface limits the resource manipulations that can be identified and the safety policies that can be enforced. For example, if we place the platform interface at the level of system calls, we cannot express safety policies that constrain resources that may be manipulated without using system calls, such as memory or processor usage. If the platform interface is placed at the level of machine instructions, we could describe more policies, but it would be harder to write a correct platform interface, and the analyses and transformations necessary to enforce a policy would be more complicated and expensive. Hence, we focus on platform interfaces at the level of system calls.

For Naccio/JavaVM, we are limited by our ability to deal easily with code for native methods. This means that at a minimum, the platform interface must describe how native methods in the Java API affect resources. We can either prevent an application from installing and using additional native methods, or require that the implementations of those native methods be transformed to enforce the desired policy by a Naccio implementation for the native platform. This allows us to use the same policy on both the Java classes and native methods.

For every other API method, constructor and initializer, we can decide either to describe it using a platform interface wrapper or to let it pass through checking so it is treated as part of the application. Although passing through checking is the less error-prone approach, it may be worth declaring wrappers for some methods instead to improve efficiency and clarity. In other situations, it may be useful to deliberately write platform interface wrappers to allow system code to manipulate resources without corresponding checks being performed. For example, we may wish to write a wrapper for the API method that loads a font so that safety violations are not reported when that method observes system properties to find the font. This would be dangerous, however, since attackers may be able to exploit the wrapped method to manipulate resources unexpectedly. In fact, versions of the JDK were vulnerable to an attack in which programs exploited font loading to access restricted information [24]. In general, the platform interface should not define wrappers for any procedure unless we are absolutely certain how it manipulates resources.

Figure 4 shows an excerpt from the Java API platform interface that defines wrappers for the java.io.FileOutputStream class. The RFile and RFileSystem classes correspond to the RFile and RFileSystem resources shown in Figure 2. The RFileMap class (not shown) keeps a mapping between Java file objects and RFile objects. We use the rfile state variable to keep track of the RFile object associated with a FileOutputStream. Wrappers for constructors must set this state to the appropriate value.

```
wrapper java.io.FileOutputStream
  requires RFileMap;
  state RFile rfile;

  wrapper FileOutputStream (java.io.File file)
    rfile = RFileMap.lookupAdd (file);
    if (file.exists ())
      RFileSystem.openWrite (rfile);
    else
      RFileSystem.openCreate (rfile);
    %%%   // marker for original call

  …  // Other constructors similar.

  wrapper void write (byte data[])
    if (rfile != null) RFileSystem.write (rfile, data.length);
    %%%

  …  // Other write methods similar.
```

Figure 4. Platform Interface excerpt.

The constructor shown calls RFileMap.lookupAdd to find the RFile object that corresponds to a Java file object. If the file map does not already contain a resource file for this object, lookupAdd creates and returns a new RFile object. Next, we pass this object to the appropriate RFileSystem resource operation. Since RFileSystem distinguishes between creating new files and writing to existing files, the wrapper calls java.io.File.exists to determine whether to call the openWrite or openCreate resource operation. It calls the unwrapped version of exists, so no safety checking is done. After this, the original constructor is invoked.

If the policy in use constrains opening files, checking will be performed in the openWrite or openCreate resource operation. If a violation is detected, the user will have the option to terminate execution before the original constructor opens the file.

The wrapper for the write(byte[]) method is defined similarly. The rfile is null if this output stream does not correspond to a file (e.g., if it is the standard output stream). Otherwise, the wrapper calls RFileSystem.write to reflect resource usage. Since write is not a native method, this wrapper is only necessary to improve the performance of checking. If it had no wrapper, checking would pass through to the native method (in the Sun JDK 1.1.7 implementation, FileOutputStream.writeBytes) that actually writes bytes to a file.

For Win32, a similar argument is used to determine the level of the platform interface [20]. The most convenient level for a Win32 platform interface is that of the Win32 API. The API has documented behavior, making the creation of platform interface wrappers relatively straightforward. Furthermore, the Win32 API is fully encapsulated into dynamic link libraries (DLLs), and thus it is easy to separate from user code.

6 Implementation issues

This section describes some issues involved in developing Naccio implementations for specific platforms. Our current experience is limited to the JavaVM and Win32 platforms. However, we believe Naccio implementations for most other platforms can be produced using similar techniques.

6.1 Policy generator

The policy generator analyzes a safety policy and produces a policy-enforcing platform library. For JavaVM, it produces policy-enforcing versions of Java API classes; for Win32, Naccio produces policy-enforcing versions of the Win32 system DLLs. Much of the work done by the policy generator is the same across all platforms. The differences are the format and content of the platform interface and the platform library.

Policy generation can be divided into two phases: generating resource implementations that perform the checking necessary to enforce a safety policy and creating a policy-enforcing library that calls those resource implementations as directed by the platform interface. For both, it is important to analyze the policy sufficiently to eliminate unnecessary overhead when the policy-enforcing library is used.

Resource implementations. Generating resource implementations involves analyzing the safety policy to determine which resource operations do meaningful checking and generating code that implements those resource operations. Code from safety properties is woven together to create the body of a resource operation.

A dependency analysis determines which resource operations are necessary. A resource operation is necessary if it could produce a violation, if it modifies some state that is used by another resource operation that could produce a safety violation, or if it has some visible side-effect. The necessary resource operations are then translated to produce a platform-specific implementation.

Naccio/JavaVM generates resource implementations as Java source code and compiles them using a standard Java compiler. Figure 5 shows the resource class for the RFileSystem resource description (Figure 2) generated to enforce the LimitWrite safety policy (Figure 3). Because RFileSystem was declared as a global resource, all class variables and methods are declared static. The bytes_written field introduced by TrackTotalBytesWritten is implemented by adding a class variable to RFileSystem.

The implementation of the write method consists of code from TrackTotalBytesWritten and LimitBytesWritten. The parameter to LimitBytesWritten used in the safety policy is bound in the code. The violation command in the safety property is replaced with a call to a Naccio library method and additional arguments are passed so an informative error message can be produced. The resource class implementation only provides implementations for those resource operations that are constrained by the policy. In this case, LimitWrite constrains only the delete, openAppend, openWrite and write operations, so these are the only methods implemented by the RFileSystem resource class.

```
package naccio.policy.limitwrite.resource;

public class RFileSystem {
  static long bytes_written = 0;
  final public static void openWrite (RFile file) {
    naccio.library.Check.policyViolation
      (“LimitWrite”, "NoOverwrite", "Attempt to overwrite file.");
  }
 … // Implementations of openAppend and delete similar.
  final public static void write (RFile file, long n) {
    bytes_written += n;
    if (bytes_written > 1000000)
      naccio.library.Check.policyViolation
        (“LimitWrite”, “LimitBytesWritten”, “Attempt to write…”);
  }
}
```

Figure 5. Generated Resource Class.

To be secure, Naccio must implement the generated resource classes in a way such that the transformed program cannot manipulate state associated with safety checking. For Naccio/JavaVM, we can do this simply by using new class instance variables and restricting use of the Java reflection classes so that the transformed program cannot access this state. We use platform interface wrappers to enforce the necessary constraints on the reflection classes.

Naccio/Win32 generates resource implementations as C source code that is compiled to DLLs. For Win32, protecting resource state poses a more serious challenge. The application transformer must ensure the necessary properties as described in Section 6.2.

Policy-enforcing library. The policy-enforcing library contains wrapped versions of system calls as directed by the platform interface. For performance requirements, it is important that wrappers are generated only for methods where policy-relevant work is done. A wrapper is necessary if it calls a resource operation that does useful work (as determined by the analyses done to produce the resource implementations), if it modifies some state that is elsewhere used in a meaningful way, or if it changes the behavior of the program (either by changing how the original method is called or by calling additional methods that have visible side effects). As with resource implementations, a deep dependency analysis is done to determine which wrappers can be safely eliminated.

Naccio/JavaVM produces the policy-enforcing library by modifying the Java API classes directly. We use the JOIE toolkit [2] to perform the necessary modifications. To implement a wrapper, we rename the original method by adding a unique prefix to the method name. This is necessary since the wrapped version of the method and other methods in the class library will need to be able to call the original method. The wrapper code from the platform interface is compiled into Java byte codes and inserted into the class file in place of the original method. Next, we need to ensure that the unwrapped version of the method is called by other wrapped API methods. Those methods have wrappers to account for their resource usage, so they should not call the wrapped versions of API methods since that would duplicate checking.

Constructors and native methods introduce a few complications. Since the class determines the names of constructors, we cannot rename constructors. Instead, we add an extra argument to distinguish the original constructor from any other constructors. This means when Naccio transforms library classes and needs to call the unwrapped version of the constructor, it must push an extra argument on the stack and change the type descriptor of the constructor it calls. Since application code always calls the wrapped constructor, there is no need to alter application classes.

For native methods, we cannot change the method name since the JavaVM will not be able to find the native method implementation. Instead, we introduce a new method that implements the wrapper and calls the original native method. This means we need to replace calls to the native method in application and unwrapped library code with calls to the new wrapped method instead. An alternative would be to rename the native method and modify the VM so that it can still map the new name to the correct native method. This would eliminate the need to replace wrapped native method names in application classes, but would not be portable across different VM implementations.

The modified classes are written to a new directory so that they can be selected at runtime by setting the CLASSPATH appropriately. If we wish to support multiple policies running in the same VM, we also need to globally rename all classes in the API to include a unique package name so that they can be identified (e.g., java.io.File becomes policy248.java.io.File). To rename classes consistently, all classes in the API must be rewritten. If applications that enforce different policies share objects that are instances of API classes, a type error will result. The problem of sharing objects between applications enforcing different policies is a complex one and will be considered in future work.

For the Win32 implementation, the policy generator produces policy-enforcing versions of the library DLLs. The policy-enforcing versions of the library DLLs contain export table entries for all functions in the original DLL. These entries identify wrapper routines that perform the necessary checking and call the library API routines using the original DLLs. Entries for which no wrapper is necessary can be implemented simply as forwarding pointers. The Win32 loader will substitute the original API call so there is no runtime overhead associated with unwrapped methods.

Policy description file. The other output of the policy generator is the policy description file. This file contains transformation rules that compactly describe the changes the application transformer must perform. It contains a rule that identifies the location of the policy-enforcing library. Rules may also direct the application transformer to rename wrapped native methods, and to modify the application to call resource initializers before execution begins and to call finalizers just before execution terminates.

6.2 Application transformer

The application transformer reads a policy description file and transforms an application accordingly. In addition, it must ensure that the low-level code safety properties necessary to ensure the integrity of the checking are enforced. The application transformer is mostly platform dependent, since it deals with low-level issues of transforming an object file.

For Naccio/JavaVM, the application transformer examines an application class to determine which classes it uses, and recursively examines those classes to determine all class dependencies. Classes that are not part of the Java API (that is, they are not described by the platform interface) are added to the classes to be transformed.

The main change the application transformer must perform is to ensure that the correct policy-enforcing library classes are used. If we are running the application in its own VM, the application transformer simply sets the CLASSPATH so that the modified classes are found before the standard Java API. For many policies, there is no need to alter the application class files and hence, there is no load time cost associated with enforcing the policy. The only cost is the run-time overhead required to do the actual safety checking for a particular policy. For some policies, the application transformer still needs to make other changes to the application classes such as renaming wrapped native methods or inserting calls in the main method to initializers or finalizers. All these changes can be performed by simple class file modifications.

If we wish to support running the application in a VM with other active policies, we need to use a version of the policy-enforcing library with renamed classes. One possibility would be to do this at run-time using namespace management. Wallach *et al.* describe how a Java ClassLoader could be modified to use this technique to hide system classes or interpose implementations with extra security checking [22]. Instead, we modify the class file at load time by replacing class names directly. The Java class file format makes renaming classes simple and efficient. All class names are given in the constant table found at the beginning of the class file. We replace class names of library files with the corresponding policy-enforcing library class name.

An advantage of renaming classes statically is that once the application has been modified it can be run repeatedly without further modification. Also, it means we are not tied to a particular Java environment. The disadvantage is that we need to be careful to make sure dynamic class loading loads the correct policy-enforcing library classes or transformed application classes. We do this by writing platform interface wrappers for the API methods that load classes dynamically. These wrappers either ensure that only policy-enforcing classes are loaded, or they run Naccio/JavaVM to transform new classes before they are loaded. Similarly, we use platform interface wrappers to prevent applications from using reflection classes to directly access methods and fields of platform library and resource implementation classes.

We use the Java byte code verifier to ensure the necessary low-level code safety properties. By verifying the application classes before they are transformed, we can ensure that an application is not able to indirectly call the unwrapped methods or manipulate resource implementations since these actions would be detected as violations by the byte code verifier. We also run the byte code verifier on the modified classes after the transformations. This step could be eliminated if load time efficiency is a priority, but is useful for detecting bugs in the transformer.

For the Win32 implementation, the DLL interface provides a convenient point at which to perform the redirection of platform calls to their policy-enforcing wrapper. For DLLs linked implicitly at load time, a simple change to the DLL name in an application's import table will redirect all calls to a policy-enforcing version of the DLL. For DLLs loaded explicitly at run-time, a wrapper for the LoadLibrary API function (which is always linked implicitly) can transparently substitute the appropriate policy-enforcing version of the DLL.

Naccio/Win32 must enforce the necessary low-level code safety properties to prevent self-modifying code (which could be used to jump directly to system DLL routines thereby circumventing safety checks) and access to protected memory (such as storage used to implement resource state). We use a limited version of software-based fault isolation [21] to ensure that all jumps remain within the application's code segment. For single-threaded applications, we can take advantage of read-only pages to provide the necessary memory safety. Since access permissions are turned on and off using system calls, we can write platform interface wrappers that limit a program's ability to change memory access permissions so that only Naccio checking code can modify the protected state. We are currently investigating methods that can efficiently provide the necessary guarantees for multi-threaded applications.

Since these protections must be applied at the level of machine instructions, the implementation of low-level code safety is dependent on the processor architecture. Our current implementation supports the DEC Alpha architecture, due to the public availability of tools for binary code modification on that architecture, including ATOM [19] and SPIKE [3]. The rest of the Win32 implementation, in particular the platform interface and policy generator, is portable across different Win32 processors.

7 Results

This section reports on results using Naccio/JavaVM to enforce several safety policies. We can view the costs of using Naccio in terms of the one-time costs associated with generating a new policy and with preparing an application to enforce a particular policy, and the overhead to enforce a policy when the application executes. While we are willing to accept high policy-generation costs, it is important that the application preparation and run-time costs are low.

7.1 Sample policies

An important advantage of Naccio's general mechanisms for defining safety policies is that a wide range of policies can be enforced. Naccio can enforce policies outside the scope of those considered by traditional systems, as well as policies that are more precise than traditional systems can support. For our experiments, we use the following six policies.

LimitWrite. The LimitWrite policy (Figure 3) prevents applications from altering any existing file and places a one million-byte limit on the amount of data that may be written to the file system.

LimitNetwork. The LimitNetwork policy limits the hosts to which an application may connect, constrains how much bandwidth may be used, and limits the total amount of data that may be transferred using the network.

Traditional code safety systems support policies that restrict network use by limiting the hosts to which the application may connect. With Naccio, we can easily express such policies by placing constraints on the resource operation associated with opening a connection. Merely restricting the hosts to which applications may connect, however, does not provide sufficient protection from denial-of-service attacks or buggy programs. Naccio can be used to define properties that limit the total amount of data an application may send over the network in a way similar to how the LimitBytesWritten property (Figure 3) limits the amount of data that may be written to files.

A more useful property limits the rate at which data is sent and received. A bandwidth limit is useful if, for example, we want to run a stock ticker applet at the same time as a video conferencing program and ensure that the stock ticker does not consume too much bandwidth at the expense of video quality. To limit transfer rates, we add resource fields that keep track of the start time for the current time quantum and the number of bytes sent during this quantum. For each send or receive, we check if the current time quantum has expired. If it has, we reset the transfer counter and start a new time quantum. Then, we increment the value of the number of bytes transferred in this quantum. If the limit would be exceeded, a violation is raised. A more useful version of this policy (enforceable using Naccio, but not used in for the performance results), splits and delays network sends to conform to the desired bandwidth limit instead of raising a violation when it would be exceeded.

Paranoid. The Paranoid safety policy includes all properties of the LimitWrite and LimitNetwork as well as properties that limit what system properties can be observed, limit how many different files may be touched, and allow a program to touch either the file system or the network, but not both. Paranoid also prevents an execution from creating windows, observing keyboard or mouse events, or manipulating threads.

JavaApplet. The JavaApplet policy duplicates the policy HotJava 1.1.5 enforces on untrusted applets. Naccio can be used to mimic any JDK policy by writing a safety policy that calls the security manager check methods at the same execution points as the Java API would call them. For better performance, we mimic the HotJava policy by moving the checking code from AppletSecurity security manager into the safety policy and making the few changes necessary to convert Java code into safety policy actions. JavaApplet disallows reading, writing and observing files except as permitted by access lists in the user's configuration file. It only allows network connections to the originating host. (For our experiments we set the originating host using a command-line definition.)

Null. The simplest safety policy is the Null policy. It places no constraints on resource usage, and is included as a baseline. To generate the Null policy, we modify the policy generator so that it removes code related to calling security manager checks from all API classes.

TarCustom. All the safety policies we have seen so far are general enough to be part of a standard policy library. Naccio is flexible enough that we can also define policies precisely targeted to the expected behavior of a particular application. While we do not expect typical users would go to the trouble of constructing an application-specific policy, it would be reasonable to expect application developers to include a precise safety policy as part of a distribution. Although we cannot depend on malicious attackers to provide appropriate safety policies, it is reasonable to expect trustworthy developers to provide an application-specific policy that protects users from bugs. A system administrator installing the application would examine the safety policy and combine it with an organizational policy.

The TarCustom policy is designed specifically for the tar archive utility. It instantiates several properties specifically targeted to the tar application, as well as some general properties, such as the NoNetwork property that disallows all network use. It includes a property that allows one file with a name ending in .tar to be overwritten if the c flag is used to create an archive, but allows no other files to be overwritten. TarCustom also limits the number of bytes written at all execution points to a function of the number of bytes read, and restricts files that may be read during the execution to those listed on the command line.

In addition to offering protection from malicious or buggy implementations, a precise application-specific policy provides protection from user mistakes. For example, executing tar -cf * using a policy-enforcing version of tar results in a policy violation. With the original application it would replace the first file in the directory with an archive of all other files.

Policy	Resource Operations	Wrapped Methods	Size (KB)	Time (min:sec)
Null	0	17	20	1:08
LimitWrite	3	30	86	1:59
LimitNetwork	14	31	103	2:12
Paranoid	51	130	320	5:08
JavaApplet	46	114	300	5:51
TarCustom	23	107	268	4:42

Table 1. Costs of generating policies. The 17 methods wrapped for the Null policy are the methods all policies must wrap to protect dynamic class loading and reflection. Time is the clock time to generate the policy not including not include time required to remove security manager calls from the Java API since that can be done once for all policies. For all our experiments, we use Sun's JDK 1.1.6 with no JIT compiler on a Pentium II/300 running Linux 1.3.

7.2 Generating policies

Table 1 shows the time required to generate each sample policy and the size of the policy-enforcing library. Although complex policies take several minutes to generate, this is believed to be acceptable since policy generation is an infrequent task.

Policy generation time is dominated by the dependency analysis needed to determine which resource operations and wrappers are necessary and the time required to parse, transform and rewrite the Java library classes. The current implementation is extremely inefficient, and it would not be difficult to produce a better implementation that reduces these times dramatically.

The size of the policy-enforcing library depends on how much of the API needs to be modified and how many resource operations are required. In the worst case, Naccio would need to copy the entire API. For the normal case, however, only a subset of the API classes need modifications and Naccio need only generate those classes. If we are not renaming classes, we simply generate the classes that need to be modified, and the runtime will find the modified class if it exists, or the original class if it does not. For the version of the policy with renamed classes, we need to rewrite all API classes so the size of the policy-enforcing library is approximately the size of the Java API (about 9 megabytes).

7.3 Transforming applications

The time required to transform an application is important, since users experience it every time a new program is run with a safety policy. The prototype implementation transforms about 15KB of application classes per second. Most of the time is spent parsing class files. This performance would be unacceptable in most actual uses but it could easily be improved. In particular, we can reduce the overhead of application transformation to nearly zero by integrating it into the byte code verifier. The actual work needed to transform an application is limited to some simple string replacements in the constant pool at the beginning of each class file and for some policies inserting a few calls to initializers and finalizers into the main method.

7.4 Execution performance

Assuming the policy generation and application transformation costs are acceptable, the most important cost of enforcing a safety policy is the run-time overhead experienced when the program is run. For our tests we use the following benchmarks:

- Blast – a toy application that tests if a file exists and observes several system properties in a loop that executes 100,000 times. This benchmark requires a large amount of security checking relative to the amount of real work.
- Tar – an implementation of the tar file archiving utility from www.ice.com. It is run to create a new archive of a directory tree containing 10MB in 3152 mostly small files.
- Socket – an application based on the java.net.Socket example in *The Java Class Libraries* [1]. In a loop that executes ten times, it opens a socket to an allowed host, sends 50,000 bytes in 10-20 byte blocks, and closes the socket.

Figure 6 compares the performance overhead to enforce the Null and JavaApplet policies on each application using Naccio and the JDK. The Null policy is used as a baseline since it does no security checking. We compare it to the JDK-Null policy, a security manager where each check method has an empty body. The overhead of calling the security manager check methods averages a few percent (up to five percent for the Tar benchmark). This overhead varies depending on the API methods called by an application, but is incurred when we use JDK security mechanisms regardless of the actual policy.

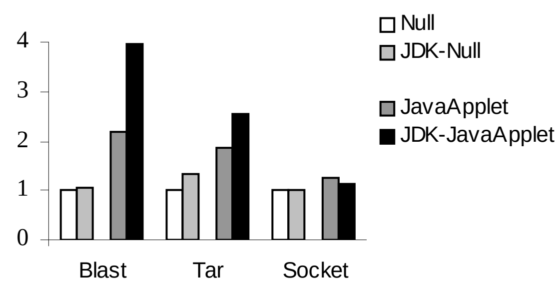

Figure 6. Performance. Each bar indicates the execution time using the given policy divided by the run-time for the Null policy. Timings are clock time average of four trials.

We compare the Naccio JavaApplet policy to the JDK-JavaApplet policy. This is the security manager used by HotJava modified to enforce the same policy on applications as it does on applets (by changing the return value of one function). To avoid any security violations, we set the acl.read and acl.write properties to allow the necessary reading and writing, and set the originating host to allow the network connections made by the Socket benchmark.

The relative security overhead varies greatly according to the type of application. The overhead to enforce the JavaApplet policy on the Blast and Tar benchmarks is significantly lower using Naccio than using the JDK. For the Tar benchmark, Naccio enforces the policy with 84% slowdown, compared to 153% slowdown using the JDK to enforce the same policy.

For the Socket benchmark, using Naccio to enforce JavaApplet requires slightly more overhead than using the JDK. This is because the Naccio implementation of the policy must create RNetConnection objects to represent network connections whereas the JDK security manager uses the Java Socket objects directly. The run-time of the Socket benchmark is dominated by network sends. Since the Null and JavaApplet policies place no constraints on sending once a socket is open (and the JDK cannot impose any such constraints), the effect of different enforcement mechanisms is less significant on its overall performance.

Naccio offers two performance advantages over the JDK security manager approach. Whereas the JDK has to determine which security manager to use and whether or not to apply a policy by examining the ClassLoader stack at run-time, Naccio makes all policy decisions statically when the policy is generated and the application is transformed. This effect is clear when we compare the results using Naccio to enforce the Null policy to those using JDK security mechanisms with an empty security manager.

The other performance advantage of our approach is that checking overhead is incurred only for API calls that are constrained by the actual policy. This becomes clear when we compare the relative costs of enforcing different policies on different application. Figure 7 shows the relative enforcement overhead of the LimitFile, LimitNetwork, Paranoid and TarCustom policies on the benchmarks. Constants used in the Naccio policies (such as the write limit and network rate) are set high enough so that no violations are detected running the benchmarks.

There is no meaningful comparison to the JDK for these policies, since it is impossible to enforce them using the JDK. Nevertheless, Naccio enforces most of these policies with less overhead than necessary to enforce the JavaApplet policy using the JDK. Enforcing the Paranoid and TarCustom policies on the Tar benchmark requires slightly more overhead than enforcing JavaApplet using the JDK.

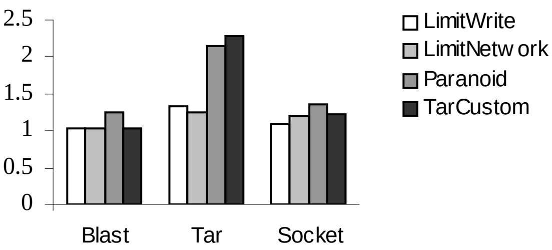

Figure 7. Policy enforcement costs. Each bar gives the execution time for running the benchmark enforcing the given policy divided by the execution time for running the benchmark enforcing the Null policy.

The execution cost varies according to the complexity and ubiquity of the policy. Enforcing a policy that places no meaningful constraints on the application has negligible overhead. For example, the slow-down for enforcing LimitNetwork on the Tar benchmark is minimal. Since the only safety checking is associated with network-related methods that are never called by tar, there is no significant overhead necessary to enforce the policy.

On the other hand, enforcing a complex and ubiquitous policy has a significant performance penalty. Enforcing the policies that limit the number of bytes that may be read or written on the Tar benchmark illustrates the worst-case scenario. Most of the work done by the application is in executing a large number of fast system calls that need to be checked to enforce the safety policy. For each write, TarCustom increments a state variable that keeps track of the number of bytes written and then compares this value to a function of the number of bytes read.

The performance disadvantage of using Naccio is the overhead required to create and maintain abstract resource objects. In cases such as the Socket benchmark with the JavaApplet policy, this overhead may outweigh the performance benefits of statically determining policies. Perhaps a better policy generator could reduce this overhead by optimizing out resource objects in cases where it is possible to derive the necessary information from the unmodified Java object. In other cases, the extra state information kept in resource objects improves checking performance considerably. For example, the checks relating to file readability in the JavaApplet policy can be performed more quickly because the absolute pathname of a file is stored in its corresponding RFile resource object. The JDK implementation needs to recompute this path for every check that uses it.

Unlike traditional code safety systems where much of the overhead is incurred irrespective of the safety policy being enforced, with Naccio the run-time overhead is directly related to the checking needed to enforce a particular safety policy. Naccio can provide the extra flexibility needed to enforce a wide class of policies without suffering a performance penalty when simpler policies are enforced.

8 Related work

This section surveys relevant work in low-level and high-level code safety. Work in low-level code safety provides the foundation necessary to support Naccio's high-level safety mechanisms. We survey how other high-level code safety approaches express and enforce policies. The most closely related work is the work on enforcing resource limits, execution monitoring and the Ariel project.

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 [8]. Without control flow safety, an attacker could construct a program that jumps directly to the body of a system call bypassing any restrictions or checking code. Without memory safety, a program could modify its own code and the data used in safety checking. Approaches that offer the most promising options for providing the low-level code safety needed by Naccio include verification and software fault isolation.

Verification. One way to ensure low-level code safety is to verify the needed properties of an object file before allowing it to execute. If the property cannot be proven, the code is rejected.

Java uses a byte code verifier [25] to provide low-level code safety. Before loading a class, the verifier performs data-flow analysis on the class implementation to verify that it is type safe and that all control-flow instructions jump to valid locations. Naccio/JavaVM relies on the Java byte code verifier to provide low-level code safety.

A more ambitious verification approach is Proof-Carrying Code (PCC) [11]. PCC combines a program with a proof that the program satisfies certain properties. Before installing the program, a certifier verifies the proof. Proof generation may be complex and time-consuming, but verification should be simple and efficient. In theory, PCC techniques can be used to verify both low-level and high-level code safety properties. In practice, they are limited by automatic proof-generation technology, and have been used most effectively to verify low-level safety properties [12].

Software fault isolation. Instead of proving that arbitrary code has desired properties, it is usually easier to transform code in a special-purpose way so that it is guaranteed to have the desired properties. Software fault isolation (SFI) [21] introduced this approach to provide low-level code safety. It uses bit masks to ensure that memory operations and jumps access only the correct memory ranges. Omniware [10] uses SFI to provide low-level code safety in a mobile code system. Software Fault Isolation has been implemented on other platforms including Alpha [17] and Intel x86 [18]. Naccio/Win32 uses software fault isolation to prevent application code from jumping directly into system code.

8.2 High-level code safety

Provided low-level code safety is in place, we can employ mechanisms for high-level code safety knowing that they cannot be circumvented by forged pointers or arbitrary jumps. A number of high-level code safety techniques have been invented. We describe those most relevant to our work, focusing on how they describe policies, what policies they can enforce, and the enforcement mechanisms they employ.

Safe-Tcl. Safe-Tcl [13] is a version of the Tcl scripting language designed as a safe platform for running untrusted scripts that control the behavior of trusted containing applications. Safe-Tcl implements a safety policy by hiding commands from an untrusted script. A safety policy can be defined to allow the untrusted code to access hidden commands through aliases that may do checks before calling the unsafe command. One limitation of Safe-Tcl policies is that only a small fixed set of commands can be controlled. Naccio's use of policy-enforcing libraries is similar to Safe-Tcl's use of command aliases.

Java. The Java run-time environment [5, 7] provides high-level security by allowing access to system resources only through the Java API. Functions in the API are implemented to call safety checks before certain system calls are executed.

In implementations before JDK 1.2, the safety checks are performed by `check` methods of the `SecurityManager` class. The `SecurityManager` acts as a reference monitor [9], enforcing a particular safety policy by controlling access to system calls. If the safety policy disallows a call, a security exception is raised before the system call can be executed.

New safety policies can be defined and enforced by writing a `SecurityManager` subclass. The scope and precision of policies, however, is limited by where the system libraries call security manager methods and by how much information is passed to the check method. For instance, the constructor for `FileOutputStream` calls the `checkWrite` method before opening a file, but the `write` method does not call any security manager method. Hence, one can implement an arbitrary security policy on what files may be written by changing the `checkWrite` method, but can place no constraints on the amount of data that may be written to a file once it has been opened.

Although one could imagine solving this problem by simply adding more security manager calls to the Java API, the performance penalty associated with this straightforward solution would be unacceptable. The problem is that the security manager method must be called regardless of whether or not the safety policy in use constrains it. This overhead is acceptably small in situations where it is used for expensive operations like opening a file or establishing a network connection. However, it would be unacceptable to require it for every system call, especially inexpensive, frequent ones like writing a byte to a file.

By requiring the cost of a security manager check regardless of whether or not the safety policy places any constrains a particular API call, the JDK security mechanisms limit the API methods that call security manager checks, and hence the range and precision of safety policies that can be enforced. Because the security manager not usually known at compile time, no matter how good compilers get they are unlikely to be able to optimize out calls to security manager checks. Naccio avoids this problem, since static analysis of safety policies ensures that wrappers are applied only to methods that may manipulate a resource in a way constrained by the selected safety policy.

Much of the work in Java security has been directed at providing greater flexibility as to which policy is applied to a particular class. The Java class loading mechanism offers opportunities here, since each run-time class is associated with a ClassLoader that can be used to determine safety properties. A particular problem is distinguishing between applet code, which should be limited by a particular safety policy, and system code, which may be granted extra privileges. One technique for dealing with this is stack inspection [22]. System classes are permitted to enable privileges, but they are enabled only for inner calls that remain inside the system code. When a privilege is required for an operation, the run-time system examines the stack to determine if the privilege was enabled by some system code, and if all classes on the stack frame after it are system classes.

Stack inspection can be implemented by transforming code to pass an extra parameter with each call that encodes which privileges are enabled [23]. By making all policy decisions statically, Naccio avoids the need to make any of these distinctions at run-time. This eliminates the complexity and vulnerability associated with determining the appropriate policy to apply to a particular class at run-time.

The Java security model continues to evolve with new Java releases [7]. JDK 1.2 introduced a more flexible security model in which the class loader can assign a different security policy to each class as it is loaded and stack inspection is used to determine what privileges are enabled. JDK 1.2 also introduced the AccessController as a more abstract and flexible alternative to the SecurityManager. Instead of calling a particular check method, implementations call an AccessController method that checks if the necessary permissions are enabled. While these changes have made it easier for vendors and users to implement different safety policies, they have not expanded the scope or improved the precision of policies that may be enforced since they are still limited by where the API calls safety checks, and are trapped because Java security mechanisms offer no way to eliminate unnecessary checks depending on the policy in use.

Resource limits. Most work on resource limits has been done in the context of operating systems instead of application-level code safety. Here, we consider work on applying resource limits to Java programs.

JRes is a resource management interface for JavaVM programs [4]. It supports per-thread accounting for heap memory, CPU time and network usage. Limits can be placed on the amount of a particular resource a thread may consume, and callbacks are invoked when a limit is exceeded. In JRes, policies are described by application calls to methods that set fixed value limits on a predefined set of resources. Many policies that Naccio can enforce could not be defined using JRes because they depend on resource manipulations not constrained by JRes or they place more complex constraints on resource usage than a fixed limit (e.g., a rate or a function of other resource usage).

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

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

Execution monitoring. Schneider defines EM, a class of enforcement mechanisms that enforce security policies by monitoring a target system and terminating an execution immediately before the policy would be violated [16]. Enforcement mechanisms in class EM can only enforce security policies that are safety policies. That is, policies 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 just monitoring an execution. However, if we place a few restrictions on the platform interface, safety property definitions, and static analyses done by the application transformer, then Naccio can be viewed as an execution monitor in class EM.

The necessary restrictions are that platform interface wrappers and safety property checking code only modify state invisible to the application, perform side-effect free computation guaranteed to terminate, and issue violations. Further, all platform interface wrappers must call the original method with the original arguments on every execution path that does not report a violation. In addition, the application transformer may not do any interesting static analyses or transformations to the code.

Policies that do not satisfy these restrictions can change the behavior of the program in more fundamental ways and are harder to classify. For example, we have defined a policy that enforces a soft bandwidth limit. Instead of issuing a violation when the bandwidth limit would be exceeded, it splits and delays network sends to stay within the requested limit. Examples of other reasonable policies enforceable by Naccio but not by execution monitoring include a policy that adds warning strings to the titles of windows created by an untrusted execution and a policy that redirects all network sends to a local file.

Schneider suggests techniques for using finite-state automata to express safety policies enforceable by execution monitoring. Úlfar Erlingsson has developed Security Automata SFI Implementation (SASI) [6], a system that enforces policies defined using automata by inserting code in program executables similarly to what is done by Naccio. 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 abstract resource objects. The transformations and analyses SASI has to do to enforce a policy are more complex than those that must be done by the Naccio application transformer, so we expect the application preparation costs will be higher with their approach. The main advantage of Naccio over SASI is that Naccio offers a convenient, platform-independent way of defining policies. By describing policies at a lower level, SASI can define and enforce policies that cannot be enforced by Naccio such as those that constrain memory accesses or the structure of the code. However, SASI cannot describe or enforce policies that modify the actual behavior of the program (such as the policy that alters network sends to conform to a requested bandwidth limit).

Ariel Project. The Ariel project describes policies using a declarative language and enforces policies by inserting code in Java classes [14]. The transformations done by Ariel to enforce a policy are similar to those done by Naccio/JavaVM. Policies are described as access constraints that prevent the creation of objects or invocation of methods based on a predicate. Because of the declarative nature of policy descriptions, Ariel is unable to describe behavior-modifying policies that can be described using Naccio's mechanisms. Policies are described at the level of the Java API so they are not portable across platforms, and writing a policy that constrains writing would require placing constraints on all methods that may write to a file (although they are working on techniques that allow classes and methods to be grouped [15]).

9 Conclusion

Our results demonstrate that it is possible to support a large class of interesting and useful safety policies without sacrificing performance or convenience. We have presented a system architecture that supports the platform-independent description of a wide range of safety policies in terms of the abstract resource manipulations. A platform interface describes how system calls affect those resources.

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

Acknowledgements

The authors thank John Guttag for his advice on and support of this work and careful reading of drafts of this paper; John Chapin, Drew Dean, Úlfar Erlingsson, Steve Garland, Brant Hashii, Daniel Jackson, Ulana Legedza, Greg Morrisett, Andrew Myers, Raju Pandey, Fred Schneider and David Wetherall for their comments on the work and/or suggestions for this paper; Geoff Cohen for making JOIE available and answering all our questions; and Robert Cohn, David Goodwin, P. Geoffrey Lowney and Dan Scales for help with ATOM and SPIKE. This research is supported in part by DARPA contract F30602-96-C-0J0J, monitored by the USAF Rome Laboratory, and by DARPA contract N66001-96-C-8522, monitored by the Office of Naval Research.

Availability

The examples used in this paper are available at http://naccio.lcs.mit.edu/.

References

[1] Patrick Chan, Rosanna Lee, and Doug Kramer. *The Java Class Libraries, Second Edition, Volume 1.* Addison-Wesley, 1997.

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

[3] R. Cohn, D. Goodwin, P. G. Lowney and N. Rubin. *Spike: An Optimizer forAlpha/NT Executables.* USENIX Windows NT Workshop. Seattle, August 1997.

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

[5] Drew Dean, Edward W. Felten, Dan S. Wallach. *Java Security: From HotJava to Netscape and Beyond.* IEEE Symposium on Security and Privacy. May 1996.

[6] Úlfar Erlingsson, personal communication, March 1999.

[7] Li Gong, Marianne Mueller, Hemma Prafullchandra. *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2.* USENIX Symposium on Internet Technologies and Systems, Dec 1997.

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

[9] Butler Lampson. *Protection.* Fifth Princeton Symposium on Information Sciences and Systems, March 1971.

[10] Steven Lucco, Oliver Sharp and Robert Wahbe. *Omniware: A Universal Substrate for Web Programming.* WWW4, 1995.

[11] George C. Necula and Peter Lee. *Safe kernel extensions without run-time checking.* OSDI '96.

[12] George C. Necula and Peter Lee. *The Design and Implementation of a Certifying Compiler.* PLDI '98.

[13] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. *The Safe-Tcl Security Model.* http://www.scriptics.com/people/john.ousterhout/safeTcl.html.

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

[15] Raju Pandey, personal communication, March 1999.

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

[17] Scott M. Silver. *Implementation and Analysis of Software Based Fault Isolation.* Technical Report PCS-TR96-287, Darthmouth College, June 1996.

[18] Christopher Small and Margo Seltzer. *MiSFIT: A Tool for Construction Safe Extensible C++ Systems.* Third Conference on Object-Oriented Technologies and Systems, 1997.

[19] Amitabh Srivastava and Alan Eustace. *ATOM: A system for building customized program analysis tools.* PLDI '94.

[20] Andrew R. Twyman. *Providing Flexible Code Safety for Win32.* MIT Master of Engineering Thesis Proposal. Feb 1999.

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

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

[23] Dan S. Wallach and Edward W. Felten. *Understanding Java Stack Inspection.* IEEE Security and Privacy. May 1998.

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

[25] Frank Yellin. *Low-level Security in Java.* WWW4 Conference, Dec 1995.