# System Call Monitoring Using Authenticated System Calls

Mohan Rajagopalan, *Member*, *IEEE*, Matti A. Hiltunen, *Member*, *IEEE Computer Society*, Trevor Jim, and Richard D. Schlichting, *Fellow*, *IEEE*

**Abstract**—System call monitoring is a technique for detecting and controlling compromised applications by checking at runtime that each system call conforms to a policy that specifies the program's normal behavior. Here, we introduce a new approach to implementing system call monitoring based on authenticated system calls. An authenticated system call is a system call augmented with extra arguments that specify the policy for that call, and a cryptographic message authentication code that guarantees the integrity of the policy and the system call arguments. This extra information is used by the kernel to verify the system call. The version of the application in which regular system calls have been replaced by authenticated calls is generated automatically by an installer program that reads the application binary, uses static analysis to generate policies, and then rewrites the binary with the authenticated calls. This paper presents the approach, describes a prototype implementation based on Linux and the PLTO binary rewriting system, and gives experimental results suggesting that the approach is effective in protecting against compromised applications at modest cost.

**Index Terms**—Intrusion tolerance, operating systems, security policy, sandboxing, compiler techniques.

## 1 Introduction

Computer systems have long been subjected to attacks that involve either altering existing code to take malicious actions or introducing new executables into the system that later compromise the system in some way. Worms, for example, propagate by altering program executables on their target. The executable might be a running program that the worm corrupts through a buffer overflow, an executable overwritten by a macroprogram contained in an e-mail message, or a library installed from a corrupted code repository as part of a regular software update. This wide mix of both targets and techniques make it challenging to develop countermeasures, especially those with broad applicability.

Despite the different approaches used to introduce altered code onto a machine, these attacks share one characteristic—they typically exploit the system call interface to take malicious action. It is only through this interface that compromised code can, for example, write to the disk or send a network packet. *System call monitoring* is a widely used technique that exploits this characteristic to detect compromised applications and sandbox them to minimize the damage they can cause [3], [8], [9], [11], [14], [15], [17], [20], [25], [28], [29], [30], [33], [34]. The approach is based on having a *policy* that captures an application's normal system call behavior and then halting execution if an application deviates from this normal behavior during execution. While system call monitoring by itself cannot fully protect a system, it has proven to be a valuable tool and can be used in conjunction with other techniques to make intrusions more difficult and to minimize their impact.

A simplified version of system call monitoring is illustrated in Fig. 1. The System Call Monitor (SCM) checks system calls at runtime using the application's policy, and either allows or disallows them depending on whether or not they match the policy. In the figure, the vertical line from the application through the SCM to the OS kernel on the left labeled A illustrates a call that is allowed to proceed after being checked by the SCM. The line on the right labeled B, on the other hand, illustrates a call that is blocked by the SCM because it does not match the policy, i.e., deviates from the application's normal behavior. Issues that must be addressed to realize such an approach include implementing the SCM to minimize runtime checking overhead and vulnerability to attack, determining an application's normal behavior and using that to generate a policy, and making policies available to the SCM at runtime. Note that the SCM can also be viewed as a reference monitor [2] for system calls, that is, it validates all systems calls made by a program against those authorized for the program.

This paper presents a new approach to implementing system call monitoring that uses *authenticated system calls* as its key mechanism. An authenticated system call is a system call that has been transformed to include additional arguments that specify the call's policy, information about the current execution state, and a message authentication code (MAC) that guarantees the integrity of the policy and relevant arguments. This MAC is computed using a cryptographic key that is available at runtime only to the kernel. When an authenticated call is invoked, the kernel computes an encoding of the call's runtime behavior from the arguments and other information, computes a MAC over this encoding using this same key, and compares it

- *M. Rajagopalan is with Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 94054. E-mail: mohan.rajagopalan@intel.com.*
- *M.A. Hiltunen, T. Jim, and R.D. Schlichting are with AT&T Labs-Research, 180 Park Avenue, Florham Park, NJ 07932. E-mail: {hiltunen, trevor, rick}@research.att.com.*

*Manuscript received 26 Sept. 2005; revised 6 Apr. 2006; accepted 25 Apr. 2006; published online 3 Aug. 2006.*
*For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSCSI-0133-0905.*

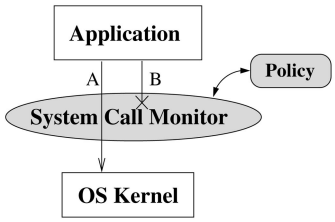


Fig. 1. System call monitoring.

with the MAC in the call. If it matches and the call passes other checks, the call's behavior complies with the policy and it is allowed to proceed; otherwise, the application is terminated.

A key aspect of our approach is that, even though the policy, extra arguments, and MAC are part of the application, a compromised application cannot successfully create a new authenticated system call or tamper with an existing authenticated call since it does not have access to the cryptographic key. This division of SCM functionality between the application and the kernel is the key contribution of this paper. Our approach contrasts with other system call monitoring approaches that either rely on user-space policy daemons [8], [9], [14], [17], [28], [33] or require large-scale changes to the kernel [3], [20], [29], [30]. In comparison with our approach, the former can have unacceptably high execution costs unless frequently used system calls are special-cased for enforcement in the kernel, while the latter results in a more complex kernel. Assuming identical policies, all of these approaches are to a first approximation comparable in the types of attacks they prevent.

The second important aspect of our approach is the automatic generation of each system call's policy and the automatic transformation of the application to replace each call with the equivalent authenticated call. This is done by a *trusted installer* program that reads the application binary, uses static analysis to determine the appropriate policy for each call, and then rewrites the binary with the authenticated calls. The use of static analysis has significant advantages over methods based on handwritten policies or policies obtained by training, i.e., recording the system call behavior of the application over a period of time. In particular, it is completely automatic, produces policies quickly, and does not miss system calls invoked by rarely used parts of the application. We demonstrate these advantages empirically by comparing our policies with those published elsewhere for the well-known Systrace system call monitoring system [25].

The primary contribution of this paper is to present a new approach to implementing system call monitoring based on authenticated system calls. This is done as follows: First, Section 2 explains system call monitoring in more detail. Section 3 then describes the details of our approach, including authenticated system calls, policy generation, installation, and system call checking. Section 4 describes a prototype implementation on Linux that uses the PLTO binary rewriting system [27] to do policy generation and installation; this section also provides experimental results, including performance overhead and evaluation of the effectiveness of policy generation. We describe potential extensions to policies in Section 5. Finally, Section 6 summarizes the contributions of the paper.

## 2 System Call Monitoring

The basic idea of constructing semantic models of a program's legitimate system call behavior in terms of sequences of system calls and monitoring departures from such models was originally proposed by Forrest et al. [15], [34]. Since then, many different system call monitors have been developed that vary in how they address the following fundamental issues related to policies:

- *Policy expressiveness.* What policies can be enforced?
- *Policy creation.* How is the policy of an application determined?
- *Policy enforcement.* Are policies enforced in the kernel, outside the kernel, or using some combination of the two approaches?

To introduce the basics of system call monitoring and how authenticated system call relates to other approaches, we consider each of these issues in turn.

### 2.1 Policy Expressiveness

In system call monitoring, each system call in a program has an associated *system call policy* that specifies properties that must be satisfied when the call is executed. The program's *overall policy* is the collection of its system call policies. In principle, a system call monitor should be able to enforce any computable policy. In practice, however, most system call monitors restrict the class of policies that they enforce for reasons of either efficiency or simplicity. The goal of this section is to give some examples to illustrate common types of policies.

The properties expressed by a system call policy can be viewed as constraints on the execution of the system call. A typical policy, for example, may require that a system call be constrained to a specific system call number (name), or must be invoked from a particular memory address in the program (call site), or both. It might also specify allowed values for the arguments, using either concrete values (e.g., "5" or "/dev/console") or patterns (e.g., "/tmp/*"). Policies of these types are used in many existing system call monitoring systems. For example, Systrace supports policies in which the system call number and the argument values can be specified, the latter using either patterns or concrete values.

System call policies can also constrain more global behaviors, such as the acceptable order of system call executions. For example, the collection of system call policies for a program might constrain the application's system call trace to be a path in the call graph. In this case, each system call policy could include a list of system calls that are possible predecessors for the given call. Policies of this type are also supported in certain existing systems [11], [15], [31], [33], [35].

While system call policies can be quite general, they do have one inherent limitation—they can only be checked when a system call is invoked. As a result, system call monitoring cannot *prevent* attacks such as buffer overflow;

once a monitor approves a system call that reads into a buffer, there is no mechanism in place to prevent the read from going past the end of the buffer. Rather, the goal of system call monitoring is to isolate or sandbox compromised applications to minimize the damage they can cause.

Therefore, in system call monitoring, a process can become corrupted through a buffer overflow or some other attack and then run freely, as long as it does not make a system call. Depending on the operating system, such a process can consume significant resources or take other actions that affect the system. For example, under Linux, a process can consume CPU cycles, access its memory space to cause paging, or even perform reads and writes to files that were memory mapped before the attack occurred. However, the process would not be able to open new files, or even write to files or network sockets using file descriptors, since those operations involve system calls.

### 2.2 Policy Creation

System call policies are meant to capture the normal legitimate behavior of a program. In practice, three techniques can be used to create policies:

- *Manual.* Policies can be written by hand.
- *Training.* Policies can be learned by examining some sample runs of the program.
- *Static analysis.* Policies can be determined through static analysis of the program.

Handwritten policies can be extremely valuable, particularly when they are written by someone with deep knowledge of the program. However, they are tedious to produce and maintain, so they are most often used selectively in combination with the other two techniques.

Most existing system call monitors determine policies through training. Training can be automatic, so policies are easy to produce and maintain. However, training by its nature does not examine all possible behaviors of the program, so policies produced through training may be overly restrictive. This is a significant barrier to the use of the system call monitor, because system administrators must either spend a lot of time checking false alarms or start ignoring alarms, including alarms for real attacks.

Relatively few system call monitors use the third technique, static analysis; Wagner and Dean [33] and Giffen et al. [11], [12] are notable examples. Static analysis is automated, so policies are easy to produce and maintain, and conservative analyses have the great advantage of eliminating false alarms. Our implementation of authenticated system calls uses conservative static analysis to produce policies.

Ideally, whatever procedure is used to produce policies, the end result would be a policy that exactly captures the acceptable behavior of the program. Of course, in practice, this is impossible. An extreme example is a root shell program, since for administrative purposes, a root shell may need to take arbitrary actions, including overwriting the entire file system. As a result, it is difficult to impose a policy that would provide meaningful protection in the event the shell program becomes compromised.

A more subtle example was explored by Wagner and Dean [33], who introduced *mimicry attacks*. In a mimicry attack, a corrupted program is able to find some sequence of system calls that is within the normal behavior of the program according to policy, but which can nevertheless cause harm. The best defense against mimicry attacks is to have more precise policies. While our work has not focused on this issue directly, we can implement all of the sorts of policies used in other system call monitors with at least comparable efficiency, and hence, provide as much protection against mimicry attacks as other system call monitors. In addition, note that other measures designed to address mimicry attacks specifically can be used in conjunction with authenticated system calls to provide additional protection [21].

### 2.3 Policy Enforcement

A system call monitor checks each system call against its policy at runtime and either accepts or rejects the call. This security-critical check can be performed in user space or in the kernel. Systems that intercept system calls in user space [18], [19] can be vulnerable to corruption by such exploits as buffer overflows when applied to programs written in unsafe languages. Such an approach may, however, be appropriate for programs written in safe languages such as Java [32] or in systems where unsafe languages are compiled with additional checking for safety [7], [23]. Systems that work with possibly unsafe binaries like ours have previously been implemented entirely in the kernel, or by a combination of a kernel hook or patch with a user-space policy daemon or monitor.

Our authenticated system calls use a novel arrangement—a kernel modification in combination with binary modifications to the untrusted user application itself, with no separate policy daemon. Instead, we use cryptographic and program checking techniques to ensure that any work done by the untrusted application regarding policy decisions is done correctly. In comparison to systems implemented entirely in the kernel, our kernel modifications are minor—a couple of hundred lines of code compared to thousands with other systems. A completely in-kernel implementation must maintain the policies and the logic for determining which policy applies to a given call; we place these burdens on the application. Note, in particular, that the exact policy for a given authenticated call is provided by the call itself. This gives us an advantage in speed and simplicity. In contrast to systems implemented with user-space policy daemons, we have the advantage of fewer context switches, leading to a very modest overhead. Not having a separate monitor process also simplifies policy checking because the operating environment (e.g., current working directory) does not have to be mirrored, some race conditions are avoided, and the monitor process cannot be killed by a compromised user application.

## 3 Using Authenticated System Calls

### 3.1 Approach Overview

Our system call monitoring technique is based on three steps: analyzing the program to generate policies, transforming the program to replace system calls with authenticated system calls, and runtime checking by the kernel to ensure that each call matches its policy. The first two steps

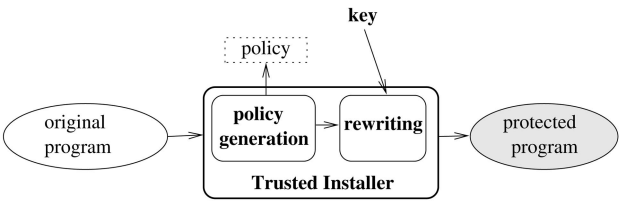


Fig. 2. Program installation.

together comprise the *installation* process, which is illustrated in Fig. 2. The program binary is read by a trusted installer program, which first uses static analysis to generate a policy that captures different characteristics of the expected behavior for each system call, and then rewrites the binary so that each system call is replaced by an authenticated system call. The arguments of an authenticated call include the arguments of the original system call plus some additional arguments, including policy information and a cryptographic MAC. The key for the MAC is specified during the installation process. The last step, *system call checking*, is illustrated in Fig. 3. At runtime, each system call is intercepted by the kernel and, after verifying the MAC using the same key as used during installation, the behavior of the call is verified against the policy. If the behavior matches the policy, the call is allowed; otherwise, the call is rejected and the executing process terminated.

Our assumptions about attacker behavior are consistent with those used for other system call monitors. We assume that applications may contain any type of vulnerability, including those that make them susceptible to attacks such as buffer overflow attacks, heap overflow attacks, and format string attacks. We also assume that the attacker may have access to the application source and binary, and that it can tamper with any part of the binary using tools such as debuggers and simulators. Finally, we assume that the MAC and policy information are visible as plain text in the binary, that the key is accessible only to the installer and to the kernel, and that it is computationally infeasible to break the key.

Our current system call policies allow properties related to the system call name, call site, control flow, and constant parameter values such as integers and string literals to be constrained. As an example, the installer might be able to derive the following logical policy for an open call in the program:

```
Permit open from location 0x806c462
  Parameter 0 equals "/dev/console"
  Parameter 1 equals 5
If preceded by the system call at 0x80a1c04
```

This policy captures the expected behavior of the call, i.e., that the call is an open from the call site at memory address 0x806c462, and that the first argument is a pointer to the string "/dev/console" and the second argument is the constant 5. Furthermore, the previous system call made by the program must have been from location 0x80a1c04. If a policy does not give a value for a parameter, then the parameter is unconstrained and any value is allowed. Section 5 extends policies to include, for example, policies

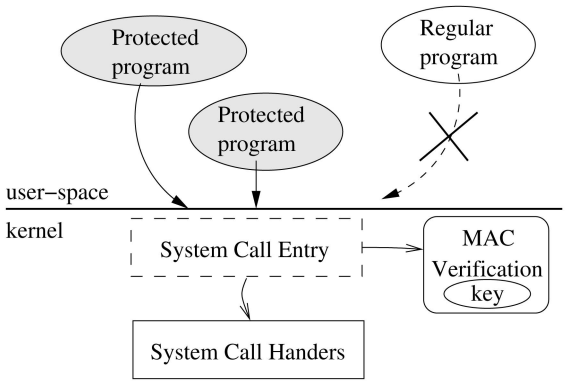


Fig. 3. System call checking.

that allow argument values to match patterns, and capability tracking policies for arguments such as file descriptors.

### 3.2 Authenticated System Calls

Authenticated system calls are used as the key mechanism for making the policy and other information needed for system call checking available to the kernel. To do this, a representation of the policy must be constructed and included as one or more additional arguments to each system call. Specific challenges in constructing such a representation include the following:

- *Policy variations.* The policy must describe which system call properties are included in the policy for a given system call. For example, some of the argument values may be constrained while others may remain unconstrained.
- *String literals.* Particular care must be taken for arguments constrained to be string literals, because the actual arguments can have unbounded length and can potentially be chosen by an attacker.
- *Global behavior.* Specifying and implementing policies that constrain system call ordering requires not only establishing relationships among system calls, but also maintaining at runtime information about system calls as they are executed.

Of course, the policy representation must also be designed in such a way that an attacker cannot modify the policy undetected. This aspect is implemented using a MAC as described in the previous section; to avoid confusion with other MACs that will be introduced, we subsequently refer to this MAC as the *call MAC*.

The need to support policy variation is addressed by constructing for each system call a *policy descriptor*, a 32-bit integer that encodes information about which properties of the system call are constrained by its policy. This descriptor uses bits to indicate whether the value of each argument is determined by the policy. It also indicates whether the control flow policy for the call is specified. The policy descriptor is then included as one of the additional arguments in an authenticated call so that it is available to the kernel for the checking phase.

Policies often require string arguments to be some constant, but their arbitrary length and the indirection that

results from the use of a pointer as the actual argument value mean that they need to be treated differently than fixed-length numeric constants. For numeric constants, including the argument value in the call MAC calculation is sufficient to detect at runtime whether it has been changed. However, a string argument is passed to the kernel as a pointer to a NULL-terminated sequence of bytes. If this sequence cannot be modified—for example, because of read-only memory protection—then it suffices to include the address of the sequence in the call MAC calculation in the same way. However, this assumption is not always valid. In such cases, we need to authenticate the contents of the string, not just its address. The difficulty here is that the attacker may replace a short string with a string that is either very long or that extends into an inaccessible portion of the address space; this could cause the checker to take excess time (denial of service) or even trigger a kernel bug.

We address this issue by creating a new *authenticated string* (AS) abstraction that is represented as the tuple `{length,MAC,string}`, where `length` is a 4 byte entry, `MAC` is a 128 bit message authentication code computed over the contents of the string, and `string` is the contents of the string. Each string constant used as an argument in a system call is transformed by the installer into an AS, with the representation being stored in a new section of the binary. The pointer to the original string contents in the argument list is then replaced with a pointer to `string` within the AS. At system call time, the kernel verifies the contents of the string using the MAC in the AS representation.

The final challenge of constraining more global behaviors requires a way to specify and check policies that relate multiple system calls. For our control flow policies, for example, it must be possible to specify a policy in which a given system call is allowed only if it immediately follows one in a certain set of possible previous system calls, as determined by the static analysis done during installation. The first thing required to relate multiple calls is a way to identify each call. While the exact address could be used, to simplify both the analysis and the implementation, we approximate system call locations by the basic block that contains the system call, as computed by the installer. (This representation also helps address specific types of attacks, as described below in Section 5.) For control flow then, the set of possible predecessor blocks is added as an extra argument to the authenticated call as part of the overall system call policy. Since this set is of arbitrary size and must not be corrupted by an attack, it is stored as an authenticated string, with a pointer being used as the actual argument.

Checking control flow policies or other global policies at runtime often requires maintaining a *policy state*. For example, for control flow policies, this state would include the basic block of the most recently executed system call. While it would be reasonable to maintain a small fixed-size constant such as this directly in the kernel, other global policies might require more extensive state and have a correspondingly bigger impact on the kernel. Rather than take this route, we instead embed this state in the application binary, with updates done by the kernel.

Maintaining policy state in the application is done using the idea of *online memory checkers*, where a data structure is stored in unreliable memory and a trusted checker with a small amount of reliable memory verifies the correctness of each update as it occurs [4]. As a simple example of a memory checker, we implemented a scheme for maintaining the policy state for control flow checking in a variable called `lastBlock`. The kernel maintains a single counter variable for each process initialized to 0 and stored in kernel space. The application itself is extended with two variables, `lastBlock` and `lbMAC`, where the latter holds a MAC calculated across the counter and `lastBlock`. A pointer to `lastBlock` and `lbMAC` is then passed as an additional argument in each authenticated system call. At system call checking time, the kernel computes a MAC over the counter stored in the kernel and the policy state `lastBlock` stored in the application. If this MAC matches `lbMAC`, `lastBlock` has not been corrupted and can be used in the control flow policy check; otherwise, the application is terminated. To update `lastBlock`, the kernel increments the application's state counter, changes `lastBlock` to the block of the current call passed as an argument, and calculates a new state MAC over the new values of the counter and `lastBlock`. The new state MAC becomes the new value of `lbMAC` in application space. Note that the counter acts as a nonce to ensure that an attacker cannot replay old values of `lastBlock` and `lbMAC`.

Putting this all together then, an authenticated system call extends the original system call with the following arguments, assuming that control flow is the global behavior being specified:

- *Policy arguments.* The policy descriptor and pointer to the authenticated string with the set of possible predecessor system calls.
- *Policy state arguments.* Basic block of current call, and a pointer to the `lastBlock` variable that holds the identifier of the basic block of the previous system call and the policy state MAC (`lbMAC`).
- *Call MAC.* MAC calculated over the policy arguments, policy state arguments, some additional information such as the system call number and call site, and those regular arguments of the call that are included in the policy descriptor.

Thus, there are five arguments that are added to each system call by the installer to transform it into an authenticated call. We now describe that process in more detail.

### 3.3 Installation

The trusted installer program is used by a security administrator to generate the policy for each system call in an application, and to produce an executable binary that contains authenticated system calls. To do this, the installer first reads in an application binary and disassembles it into an intermediate representation that is used for all the subsequent analysis and transformations. The final step in processing the program is to convert this intermediate representation back into the appropriate binary format and rewrite the file. The system as a whole is protected once all binaries that run in user space have been transformed to use authenticated system calls by the installer.

The bulk of the work of the installer involves performing static analysis on the program and transforming it in ways

discussed above. This includes not only replacing system calls with authenticated calls, but also transformations like replacing string arguments with authenticated strings and inserting the code infrastructure needed to implement the appropriate policy state. Actually generating the system call policy for each call is one of the results of this process. Note that one benefit of using static analysis is that the policies generated are applicable and correct across all uses of the resulting executable, including the case of shared binaries.

Before replacing system calls with authenticated calls, the installer does a number of analysis and optimization steps. For example, it uses standard compiler techniques such as data and control flow analysis, strength reduction, and constant propagation [1] to determine values for system call arguments. String constants are also transformed into authenticated strings at this point. The installer then determines the application's system call graph, which is used for the control flow portion of the policy. This is computed from the standard call graph of the program by keeping only those nodes that correspond to system calls and adjusting the edges appropriately. The resulting graph uses the basic block to identify each system call, as discussed above.

Once these two steps are completed, each system call is transformed into an authenticated call by adding the extra arguments described in the previous section. As a way to illustrate some of the details of how this is done, we describe how the installer constructs the *encoded policy*—i.e., a byte string that is a self-contained representation of the policy—that is used as the basis for computing the call MAC.

The encoded policy is built by concatenating bit representations of all the elements that go into making up the policy. This includes the system call number, the address and the basic block number of the call site, the policy descriptor, the argument values for those arguments that are constrained, the set of possible predecessors, and the address of the policy state variable. Specifically, consider the following example policy:

```
Permit fcntl from location 0x806c57b in
  basic block 1234
      Parameter 0 equals ANY
      Parameter 1 equals value 2
    Possible predecessors 1235, 2010,3012
Basic block number of previous call stored
  at 0x0810c4ab
```

For this policy, the installer constructs the byte string:

```
005c 00000013 0806c57b 0000002 081adcde
00000012 <16 byte stringMAC> 0810c4ab
```

Here, 005c is the system call number of fcntl. Next, 00000013 is the policy descriptor, the 32-bit number indicating that the call site, parameter 1, and the control flow leading to the system call are constrained by the policy. This is followed by 0806c57b, the call site (address); and 00000002, the policy-specified value for parameter 1. The next 24 bytes correspond to the control flow policy. This includes 081adcde, the address at which the predecessor set is stored; 0000012, the length of the authenticated string storing the predecessor set; and a 16 byte MAC computed on the contents of this string. The last four bytes of the encoded policy indicate the address at which `lastBlock` is stored, in this case 0810c4ab.

The installer computes a MAC over this byte string using a key provided to the installer by the security administrator at startup time. The prototype uses the AES-CBC-OMAC message authentication code, which produces a 128-bit code [16]. The installer adds the call MAC to the data segment of the binary, and adds a pointer to it as an argument to the system call.

### 3.4 System Call Checking

The kernel enforces an application's system call policies at runtime. When an authenticated system call occurs, the kernel receives the normal arguments of the system call—the system call number and the arguments to the original unmodified call—and the five additional arguments—the policy descriptor (`polDes`), the block number of the system call (`blockID`), the set of predecessors stored as an authenticated string (`predSet`), a pointer (`lbPtr`) to the `lastBlock` policy state and last block MAC (`lbMAC`), and the call MAC (`callMAC`). Furthermore, it can determine the call site based on the return address of the kernel interrupt handler. Using this information, the kernel validates that the system call complies with the specified policy using the following steps:

1. Check `callMAC`.
2. Check the integrity of each string argument specified in `polDes`.
3. Check control flow policy.

If all three checks are passed, the kernel carries out the system call; otherwise, it terminates the process, logs the system call, and alerts the administrator. Unauthenticated calls are also blocked.

The first step is implemented by constructing an encoding of the system call (the *encoded call*) using the policy descriptor and other arguments in a manner similar to the construction of the encoded policy by the installer. Specifically, the encoded call is constructed by concatenating together the values that reflect the actual execution behavior of the call: the system call number, the call site, `polDes`, argument values specified in the policy descriptor, `blockID`, `predSet`, the MAC for the predecessor set authenticated string `psMac`, `lbPtr`, and `lbMAC`. For constant numeric values, the value of the argument is used directly in the encoding, while for authenticated string arguments including `predSet`, the tuple `{address, length, stringMAC}` is used. Note that since the address points to the string in the AS representation, the 20 bytes preceding `address` contain `length` and `stringMAC`. The kernel then computes a MAC over this encoded call using the key. If this MAC matches `callMAC`, the original arguments of the system call comply with the policy, the additional arguments have not been modified, the `length` and `stringMAC` fields of the AS arguments have not been modified, and the AS arguments have not been replaced with some other AS.

The second step checks the authenticated string arguments for modification using the MACs that were calculated for each string in the installation step. The kernel

simply reads a string of size `length` from the address of the string argument, calculates a MAC over it, and compares it with the string MAC. The integrity of `predSet` is checked similarly since it is also stored as an authenticated string.

The final step is to ensure compliance with the control flow policy. This is done as follows:

1. Check whether `lastBlock` has been modified (`lbMAC == MAC(*lbPtr + counter,key)`).
2. Check if `lastBlock` exists in `predSet`.
3. Increment the in-kernel counter for this application (`counter++`).
4. Update policy state (`lastBlock = blockID`).
5. Update policy state MAC (`lbMAC = MAC(blockID + counter,key)`).

System call checking is designed so that either MAC matching or the control flow check fails if an application's behavior deviates from its policy. As already noted, the arguments to the authenticated system call are under the control of the application, which means that it might have tampered with any of them, including the policy descriptor and all the MACs, or it might have tried to construct a new authenticated system call somewhere in the heap. However, any change to the system call number, call site, policy descriptor, or values of arguments constrained by the policy would result in a change to the encoded call that is constructed by the kernel. This, in turn, would change the MAC needed to pass the kernel test. Our cryptographic assumption is that it is infeasible for the adversary to construct a matching MAC for its changes without access to the key. The same reasoning holds for the MAC matching done for authenticated strings. Hence, any attempt by the application to change the system call in a way that violates the policy will fail.

## 4 Implementation and Experimental Evaluation

This section describes our prototype implementation of authenticated system calls and gives the results of an experimental evaluation. We first give an overview of the implementation, then describe the policies generated by the installer. For comparison, we also describe the sorts of policies supported by the Systrace system call monitor. Finally, we quantify the performance impact of authenticated system calls compared to standard system calls.

### 4.1 Implementation Overview

Our trusted installer implementation is based on the PLTO binary rewriting system [27]. PLTO reads a binary executable program, disassembles the binary machine code into an intermediate (assembly language) representation, and performs static analyses and optimizations on the intermediate representation before writing out an optimized binary executable. The trusted installer uses some of the static analyses and optimizations already provided by PLTO, plus some that were implemented expressly for authenticated system calls. Descriptions of the basic PLTO analyses can be found in any standard compiler text (e.g., [1]); the remainder are outlined below.

The installer works as follows: First, PLTO is used to divide the program into basic blocks and construct the program's call graph. We added an analysis step here that determines the blocks responsible for system calls, as well as the specific system call for each such block; this step uses the facts that system calls correspond to the `int 0x80` machine instruction and that the system call number is placed in register EAX before the system call. PLTO's intermediate representation includes the address of each instruction, which provides the return address of each system call. This address in turn becomes part of the policy.

Since system calls are often made from stubs that are invoked by many blocks, the next step is to analyze the call graph to identify blocks that invoke these stubs and inline the stubs. This inlining allows a different system call policy to be used for each inlined site, rather than having just one policy for the system call in the stub itself. Once the stubs are inlined, each system call site is analyzed to determine the arguments of the call. This is done by examining the values pushed onto the stack prior to the call, and applying a standard reaching definitions analysis from PLTO. This allows each system call argument to be classified as follows:

- *String.* The address of a known string.
- *Immediate.* Some other constant, e.g., a known file descriptor, the address of a nonstring, or the address of a string whose contents are dynamically determined.
- *Unknown.* The analysis was unable to predict a value for the argument. This includes cases where the argument has several reaching definitions, as well as cases where the argument has none.

Finally, the graph giving all possible system call orderings is calculated from the full call graph, which gives all possible orderings of all basic blocks. This graph indicates which system calls can immediately precede any given system call, which becomes part of the policy.

Armed with the results of this analysis, PLTO is then used to rewrite the program as described in Section 3. The installer runs on Linux, PLTO's native platform. The policy generation portion of the installer has also been ported to OpenBSD to compare policies generated on the two platforms; this is used for the experimental evaluation in the next section.

PLTO is an optimization tool, and, as a result, it requires relocatable binaries (i.e., binaries in which the locations of addresses are marked), so that addresses can be adjusted as code transformations move data and code locations. Our installer currently inherits this requirement, although it should be straightforward to generate policies for binaries without relocation information. One impact of this restriction is that the binaries we test in the next section had to be compiled from source, since binaries shipped with standard Linux and Unix distributions do not contain relocation information. Note that our installer outputs nonrelocatable statically linked binaries, since our policies include the absolute locations of all system calls.

System call checking has been implemented in Linux by adding 248 lines of code to the kernel's software trap handler, and including a cryptographic library of about 3,000 lines of code for MAC functionality [13]. The software

trap handler is responsible for identifying the system call number and arguments, invoking the appropriate system call handler, and returning the result to the calling application. We modified the handler to verify the call MAC and perform the other checks required to ensure that the system call satisfies the required policy. We have not yet implemented system call checking in OpenBSD.

As an initial test of the effectiveness of authenticated system calls against different forms of code injection attacks, we devised a set of attack experiments. We wrote a simple program that reads in a file name and invokes the /bin/ls program on the input. The file name is read into a stack allocated buffer, which can be overflowed by an attacker to gain control of the program. We constructed several attacks that took advantage of the buffer overflow:

- One attack simulated a typical shellcode attack and tried to invoke the execve system call to start a shell, /bin/sh; this failed because the new system call was not authenticated and, hence, did not have a policy argument or MAC.
- A second attack simulated mimicry attacks by reusing authenticated system calls obtained from other applications; this failed because the calls did not conform to the call graph and call site policies.
- A final attack simulated noncontrol data attacks [6] and tried to replace the argument "/bin/ls" of the existing authenticated execve system call with "/bin/sh." This failed because the policy protected string arguments against alteration.

### 4.2 Policies

As noted in Section 2, an ideal policy for a given application would permit the system call behaviors needed for normal operation and no others. If the policy permits system calls not used by the uncompromised application (*unneeded calls*), it leaves open the possibility that such calls could be exploited by an attacker. On the other hand, if the policy omits some system calls actually used by the application (*needed calls*), it raises the possibility of a false alarm or false positive that causes the application to be terminated unnecessarily. False alarms are a significant administrative headache and barrier to use.

Our approach uses a conservative static analysis to generate system call policies, which means that they include all needed calls and thus avoid false alarms. Our policies might allow unneeded system calls though, because no static analysis is able to identify the exact set of needed calls for every program. Note, however, that unneeded calls might also appear in policies produced by hand or by training. Hand produced policies can include mistakes, for example. System calls identified through training are never unneeded, by definition, but there are still opportunities for errors; for example, policies might be obtained by training on one version of an application and operating system, and used on another. In general, policies generated by training are not portable between operating systems, or even between different versions of the same operating system, and they may need to be adjusted even when only libraries are updated.

TABLE 1
Number of System Calls in Policies

| Program | ASC policy for Linux | ASC policy for OpenBSD | Systrace policy for OpenBSD |
|---|---|---|---|
| bison | 31 | 31 | 24 |
| calc | 54 | 51 | 24 |
| screen | 67 | 63 | 55 |

In order to gather some empirical evidence regarding false alarms, unneeded system calls, and operating system effects, we ported our policy generator from Linux to OpenBSD. OpenBSD is a useful test case because it supports a system call monitor, Systrace, in its default build, and researchers have published Systrace policies for OpenBSD applications. The Systrace policies are generated through training along with hand edits. Below, we refer to policies generated by our installer as *ASC policies*.

Table 1 compares the number of distinct system calls permitted in both ASC and Systrace policies for several common Unix programs: bison, the GNU Project parser generator; calc, an arbitrary-precision calculator program; and screen, a screen manager with terminal emulation. The first column gives the numbers for the ASC policy generated on Linux, the second column the ASC policy generated on OpenBSD, and the third column gives the numbers for Systrace policies published by the Project Hairy Eyeball Web site [10]. This rough comparison illustrates two things:

- There are significant differences in the system calls needed for the same application running on different operating systems; this implies that policies for one operating system cannot simply be used on another.
- ASC policies identify system calls that are not present in the Systrace policies.

Table 2 examines the policies for bison in more detail. The table shows system calls that are permitted by the ASC policy generated on OpenBSD but not by the Systrace policy, and vice versa. Note that the ASC policy includes many system calls that are not present in the Systrace policy. We believe that most of these calls are in fact needed, and we have verified some of them by hand using a system call tracer on actual runs of applications. This means that the Systrace policy can cause false alarms.

Conversely, there are a few system calls permitted by the Systrace policies that are not allowed in the ASC policy. They break down as follows:

- **mmap.** The mmap system call is implemented on OpenBSD by invoking __syscall, a generic indirect system call function. The ASC policy correctly constrains the arguments of __syscall so that only mmap can be invoked, however. With Systrace, this indirection is hidden from users since its policy does not explicitly allow __syscall.
- **close.** The call of close is not identified by PLTO due to an unusual implementation on OpenBSD that

TABLE 2
Comparison of Policies for Bison

| System call | ASC | Systrace |
|---|---|---|
| __syscall | yes | NO |
| close | NO | yes |
| fcntl | yes | NO |
| fstatfs | yes | NO |
| getdirentries | yes | NO |
| getpid | yes | NO |
| gettimeofday | yes | NO |
| kill | yes | NO |
| madvise | yes | NO |
| mkdir | NO | yes (fswrite) |
| mmap | NO | yes |
| nanosleep | yes | NO |
| readlink | NO | yes (fsread) |
| rmdir | NO | yes (fswrite) |
| sendto | yes | NO |
| sigaction | yes | NO |
| socket | yes | NO |
| sysconf | yes | NO |
| uname | yes | NO |
| unlink | NO | yes (fswrite) |
| writev | yes | NO |

PLTO currently cannot disassemble. However, PLTO always reports when it cannot completely disassemble a binary, so that the administrator would always be aware of such a problem. To date, we have not encountered similar difficulties on Linux, PLTO's native platform.

- **mkdir, readlink, rmdir, unlink.** The Systrace system uses two generic names, fsread and fswrite, to specify sets of system calls; fsread denotes read-related system calls and fswrite denotes write-related calls. The fact that mkdir, etc., are not in the ASC policy indicates that they are unneeded system calls, but their execution would be allowed with Systrace since its policy includes fsread and fswrite.

Next, we examine the degree to which each authenticated system call is protected from alteration by its MAC. In our current prototype, the system call site and call number are always protected by the MAC, as are those arguments whose values can be determined by static analysis. It is, of course, impossible to determine all argument values using such techniques; for example, the value may be read as a user input, generated as a result of a system call, or may be unknown because of pointer aliasing. In practice, however, static analysis can determine enough values to be useful [33] and it can provide a partial policy template that can then be extended by the security administrator using dynamic profiling and application knowledge.

TABLE 3
Argument Coverage

| prog | sites | calls | args | o/p | auth | mv | fds |
|---|---|---|---|---|---|---|---|
| bison | 158 | 31 | 321 | 31 | 90 | 2 | 69 |
| calc | 275 | 54 | 544 | 78 | 183 | 2 | 109 |
| screen | 639 | 67 | 1164 | 133 | 363 | 7 | 297 |
| tar | 381 | 58 | 750 | 105 | 238 | 3 | 152 |

Table 3 provides the results of generating ASC policies for four programs: the three from above and tar, the Unix archiving program. The *sites* column indicates the number of separate system call locations in the program, *calls* indicates the number of different system calls, and *args* gives the total number of arguments (not including the system call number) from all the call sites. The *o/p* column gives the number of system call arguments that are output-only arguments, that is, the argument is an address of a structure where the kernel stores the result of the call. The *auth* column lists the number of arguments that could be determined by the static analysis done by the installer and that could be authenticated by the basic approach. These results indicated that 30-40 percent of the arguments can be protected based on static analysis and the basic approach.

In addition to these arguments, there are many others that might be protected by using extensions such as those described in Section 5. Table 3 includes statistics for two of these as well: arguments where the value can be determined using static analysis but each argument may have two or more values (*mv*), and arguments that are file descriptors that were returned previously from system calls such as open or socket (*fds*).

### 4.3 Performance

This section measures the performance overhead introduced by the system call checking mechanism. We begin with a description of results from microbenchmarks that measure the impact on individual system calls, and then measure the effect on the overall execution time for a number of programs.

Table 4 presents the overheads introduced by these techniques on a per-system-call basis. These results were obtained by executing each system call 10,000 times using a loop, and measuring the total number of CPU cycles using the Pentium processor's rdtsc instruction, which reads a 64-bit hardware cycle counter. The last two rows indicate the overhead of the measurement process itself. All of these experiments measured the overhead of authenticated calls without control flow policies. Each experiment was repeated 12 times; the highest and lowest readings were discarded, and the average of the remaining 10 readings is used in the table. Column 2 gives the number of cycles required to execute an unmodified system call on an unmodified kernel, while columns 3 and 4 show the effect of authenticated system calls. The variance was observed to be low.

TABLE 4
Effect of Authentication

| *System Call* | *Original* Cost (cycles) | *Authenticated* Cost (cycles) | *Authenticated* Overhead (%) |
|---|---|---|---|
| getpid() | 1141 | 5045 | 342.2 |
| gettimeofday() | 1395 | 5703 | 308.8 |
| read(4096) | 7324 | 10013 | 36.7 |
| write(4096) | 39479 | 40396 | 2.3 |
| brk() | 1155 | 5083 | 340.1 |
| rdtsc cost | 84 | 84 | |
| loop cost | 4 | 4 | |

The results indicate a noticeable cost for the checking mechanism, about 4,000 cycles for each call. As might be expected, however, on a percentage basis, the overhead is much more significant for simple system calls such as getpid and gettimeofday than for more complex calls like write, where the costs associated with buffering and memory accesses dominate.

To measure the effect of these techniques on the overall performance of applications, we compared the running times of nine programs and their protected counterparts (Table 5). These programs can be classified as either CPU or system call intensive, as shown in the table; the CPU-intensive programs are from the SPECint-2000 benchmark suite, while the system call intensive programs are a collection of common applications that make a large number of system calls. The programs were compiled using gcc 3.2.2 into statically linked relocatables that were then processed using our binary rewriting system, PLTO. Two types of executables were created: *unauthenticated binaries* corresponding to the unmodified program and *authenticated binaries* that use authenticated system calls with the full complement of policies, including control flow policies. We use unauthenticated binaries generated by PLTO rather than simply gcc as the baseline, since PLTO itself applies certain optimizations such as dead code elimination, basic block layout, and instruction scheduling. As a result, applying these optimizations in both cases gives the most accurate representation of the actual cost of an authenticated call. The cost of transforming the programs including PLTO optimizations ranged from 3.49 seconds for vpr to 86.17 seconds for gcc.

Our experiment consisted of measuring the time taken for each program to execute on a fixed set of inputs. The `time` utility was used to measure the time taken by each program, with the total computed as the sum of the user and system time. Each experiment was repeated four times and the average is used in the table. The results, reported in Table 6, indicate a generally modest overhead ranging from 0.73 percent to 7.92 percent.

Our final experiment studied the effect of the authentication mechanism on a multiprogram benchmark. This benchmark is similar to the Andrew Benchmark and consists of a series of tasks that perform routine operations such as file creation, directory creation, file compression, file archival, permission checking, moving files, deleting files, and sorting the content of files. Each iteration of the benchmark results in the invocation of about 12,000 system calls. Authenticated versions of several general purpose tools such as gzip, gunzip, rm, chdir, mv, chmod, tar, cat, and cp were used to perform the tasks. The execution time of the benchmark using original binaries was 259.66 seconds (with standard deviation of 1.24), while the execution time for authenticated binaries was 262.14 seconds (standard deviation of 2.12), an increase of only 0.96 percent.

It is difficult to compare the overhead of authenticated system calls with other system call monitors because each system enforces different policies. Note, however, that the total overhead of our approach is well below that of other systems, even though we do policy checking on *all* system calls, unlike, for example, Systrace and Ostia [9].

TABLE 5
Benchmark Suite

| Program Name | Type | Description |
|---|---|---|
| gzip-spec | CPU | file compression program from SPEC INT 2000 benchmark. |
| crafty | CPU | Game playing (Chess) program from SPEC INT 2000 benchmark |
| mcf | CPU | combinatorial optimization program from SPEC INT 2000 |
| vpr | CPU | FPGA circuit and routing placement from SPEC INT 2000 |
| twolf | CPU | Place and route simulator from SPEC INT 2000 |
| gcc | syscall & CPU | Gnu C compiler from SPEC INT 2000 |
| vortex | syscall & CPU | Object oriented database from SPEC INT 2000 |
| pyramid | syscall | Multidimensional database index creation |
| gzip | syscall | file compression program |

TABLE 6
Performance Overhead

| *Program* | *Original* | | *Authenticated* | | |
|---|---|---|---|---|---|
| | *Time* (secs) | *Std.* Dev. | *Time* (secs) | *Std.* Dev. | *Overhead* (%) |
| gzip-spec | 152.48 | 0.10 | 154.63 | 0.05 | 1.41 |
| crafty | 107.60 | 0.09 | 109.11 | 0.14 | 1.40 |
| mcf | 237.48 | 0.32 | 239.21 | 0.85 | 0.73 |
| vpr | 17.29 | 0.03 | 17.49 | 0.07 | 1.16 |
| twolf | 391.04 | 3.77 | 397.67 | 2.13 | 1.70 |
| gcc | 93.01 | 0.14 | 94.30 | 0.27 | 1.39 |
| vortex | 164.15 | 0.14 | 165.53 | 0.19 | 0.84 |
| pyramid | 1.01 | 0.01 | 1.09 | 0.01 | 7.92 |
| gzip | 2.83 | 0.13 | 2.86 | 0.11 | 1.06 |

## 5 Improving Policies

This section describes techniques for making policies more expressive to allow, for example, more complete argument coverage. We have not yet implemented these techniques, but we anticipate that they will all be relatively straightforward extensions to the existing system. We also discuss the issue of file name normalization, and describe a novel attack that exploits our integration of policies into the system call and how it can be addressed.

### 5.1 Argument Patterns

Many system call monitoring systems allow policies that specify that an argument of a system call should match a pattern given by a regular expression. This is particularly useful for temporary files, whose names are often computed dynamically using library functions like `mkstemp`. A typical example of a pattern is "/tmp/*." Patterns can be specified by the security administrator or could be partially automated by using static and dynamic profiling. The patterns can be stored as authenticated strings. The associated MAC checking will ensure an attack cannot modify the patterns or substitute different patterns for a system call. The pattern addresses can be passed to the kernel as additional system call arguments, and the policy descriptor can be extended slightly to allow it to specify that an argument must match a pattern.

Pattern matching could be implemented by extending the kernel to perform regular expression matching. However, our approach tries to minimize kernel modifications. An alternate approach borrows ideas from program checking [5] and proof-carrying code [24]. The idea is that the untrusted application performs the regular expression matching for the kernel, and presents the kernel with a "proof" that the argument matches the pattern. The proof acts as a hint that allows the kernel to easily verify that the argument does in fact match the pattern.

This is best illustrated by example. Suppose the pattern to match is "/tmp/{foo,bar}*baz," and the actual argument is "/tmp/foofoobaz." Then, the application could match the argument to the pattern and produce the "proof" or hint (0,3). This hint would be passed to the kernel, which would verify that the argument matches the pattern as follows. The kernel would scan down the pattern, matching the initial characters "/tmp/" of the pattern to the argument. It would then reach the left brace, indicating the choice of either "foo" or "bar." The 0 of the hint indicates that the correct choice is "foo" (a 1 would indicate "bar"). The kernel would match the "foo" to the argument. Then, in the pattern, the kernel would reach the * character, indicating a sequence of any characters. The 3 of the hint indicates that exactly three characters should be matched, so the kernel skips over the second "foo." Then, the kernel matches the "baz" in the pattern to the argument, succeeding. If the argument does not match the pattern or the hint is incorrect, the check will fail. The benefit of this approach is that most of the work is done by the application, and the kernel only needs to do a simple linear scan of the pattern and hint. This minimizes the necessary additions to the kernel.

The same idea can be used to simplify the checking of control flow policies. For example, we could force the application to calculate the predecessor of the node from the list of possibilities, and pass this in to the kernel to verify.

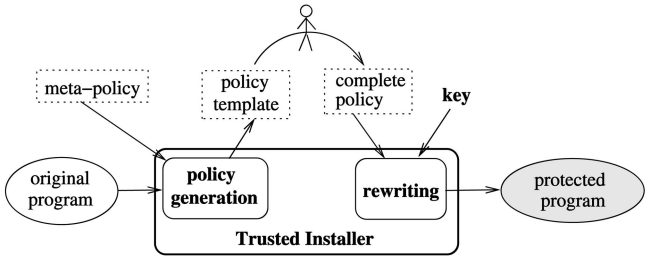


Fig. 4. Metapolicies and policy templates.

### 5.2 Metapolicies and Policy Templates

An *ASC metapolicy* is a specification that dictates how strict a policy is required for each system call. In particular, for each system call, the metapolicy indicates whether the call site must be specified in the policy and which arguments of the system call must be constrained. Compared with our basic approach, metapolicies focus on what *must be* protected for a system call rather than what *can be* protected automatically based on static analysis. Metapolicies would typically be derived from the threat level of different system calls [3] and local administrative policies.

The metapolicy is given as input to the installer along with the original program (Fig. 4). If the policy generator cannot determine all the argument values required by the metapolicy based on static analysis, it generates a *policy template* with spaces for the additional required arguments. An administrator can then hand-specify a value or a pattern for an argument based on application knowledge or dynamic profiling. The result of this is the *complete ASC policy*, which is used during the rewriting phase by the installer.

Metapolicies also play a role in extending authenticated calls to handle dynamic libraries. With dynamic libraries, system call sites for calls within the library are not known

until the library is loaded at runtime. This means that our basic approach cannot protect the call site from alteration using a MAC, as is done with statically linked binaries. In addition, arguments used by system calls in dynamic libraries are often passed as arguments to library functions, meaning that their values cannot be determined by static analysis.

Dynamic libraries are processed based on the security requirements stated in the metapolicy as follows: The dynamic libraries on a machine are installed first before the application programs. During this process, if a system call in a dynamic library function cannot satisfy the metapolicy—that is, static analysis cannot generate a complete policy—the specific function is removed from the dynamic library and set aside for static linking with application programs that require the function. Once this has been done for all system calls in the library, the functions that remain have their system calls transformed into authenticated calls in the same manner as before. Functions in this new protected dynamic library can then be loaded at runtime. Note that since a dynamic library is shared by multiple applications on the machine but a single metapolicy is used for the installation of each dynamic library, this metapolicy must be as strict as the metapolicies of the applications that use the library.

### 5.3 Capability Tracking

Another useful feature for policies is the ability to specify that an argument to a system call be based on arguments or return values of previous system calls. An example would be a policy for a read system call that requires that the file descriptor argument be a value returned by a previous open system call [30]. We call policies of this sort *capability tracking policies,* since such arguments are being used in a manner analogous to capabilities. We illustrate how the basic authenticated system call approach can be extended to support this feature using the example of file descriptor tracking.

A naive implementation of file descriptor tracking would use policy state to store the last file descriptor returned by each call to open. The policy for each read system call would specify that the file descriptor should match the file descriptor for the desired open system call. However, this ignores the fact that an open system call can be executed more than once, that more than one file descriptor returned by the open can be active at once, and that file descriptors can be reused after they have been closed.

A better approach is to store, for each open system call, a set of currently active file descriptors. The policy for each open system then adds a file descriptor to the set, while the policy for close removes a file descriptor. This involves fairly complicated data structures, so we would not use the simple policy state implementation described above, but rather a more efficient implementation based, for example, on authenticated dictionaries [22].

### 5.4 File Name Normalization

A recurring problem for system call monitors has been dealing with race conditions caused by features such as symbolic links and relative file names. For example, consider a policy that allows an application to open a temporary file, /tmp/foo. An attacker could try to exploit this by creating a symbolic link named /tmp/foo that points to /etc/passwd, and then overwriting the password file by opening and writing /tmp/foo.

To avoid this, system call monitors often use the convention that a file name in a policy must refer to the *normalized* file name, that is, the name of the file after all symbolic links have been followed. While doing normalization correctly can be complex, strategies developed elsewhere for performing this step in the kernel during system call checking [8] apply to our approach. In addition, we anticipate that it is possible to move some of the processing into the untrusted application using techniques similar to those described above in Section 3 for policy state.

### 5.5 Novel Attacks and Countermeasures

Since system call policies are compiled into the applications in our approach, it fundamentally changes the vulnerabilities of the system call monitoring system itself. In most systems, system call policies are specified in a file that is loaded by the SCM when the machine is started. At runtime, when an application issues a system call, the SCM determines the applicable policy by determining the name of the application issuing this system call (e.g., /usr/bin/login) and mapping this to the applicable policy specified in the policy file. Thus, these systems can be compromised by modifying the policy files or by replacing legitimate programs with corrupt programs whose behavior matches the original program's system call policy.

Since our approach permanently couples the applications and their policies, the above attacks are not possible in our system. However, our approach is, in principle, vulnerable to an attack that takes advantage of this fixed association. Specifically, such an attack examines multiple application binaries on the system and constructs a new application composed of authenticated system calls from these applications. We call this a *Frankenstein attack*.

A minor extension to the control flow policy implementation can prevent Frankenstein attacks. Recall that the control flow policy requires an application to execute system calls in an order consistent with its static call graph. If we ensure that basic block identifiers are unique across all programs on the same machine, the predecessor set specified in a system call policy would only match basic blocks from the same application. Specifically, a Frankenstein program would be forced into executing only the system calls of a single application, namely, the application that supplies the first authenticated system call executed by the Frankenstein program. Unique basic block identifiers can be generated, for example, by having the installer generate a short program identifier that is included as a part of each basic block identifier in the program. Note that, if actual call sites were used to specify predecessor sets, such a modification would not be possible.

## 6 Conclusions

Attacks that attempt to compromise a computer system using the system call interface are an important threat. Monitoring system calls and disallowing those that do not conform to a program's security policy is an effective

mechanisms for stopping a large class of such attacks. Essentially, a system call monitor can convert a potentially successful attack into a fail-stop failure [26] of the compromised process.

In this paper, we presented a novel approach to implementing system call monitoring based on authenticated system calls. With this approach, the policy is encoded into the application executable using a binary rewriting system, and the operating system kernel is only required to perform simple computations to verify that the actual system call satisfies the policy. This approach has been implemented using only small modifications to the kernel, without the need for heavyweight kernel data structures or the use of a user-space policy daemon at runtime. We also presented an automated approach for generating security policies based on static analysis, something that can be extended using other techniques if necessary.

We evaluated the approach on Linux and, for policy generation, on OpenBSD. In doing so, we provided measures of the effectiveness of policy generation and quantified the modest runtime impact of using authenticated system calls over unprotected ones. We also presented a number of extensions to the basic approach that can increase its effectiveness by improving the expressiveness of policies.

## ACKNOWLEDGMENTS

The authors wish to thank S. Debray who provided valuable insights on the technical issues in this paper. They also thank the referees for their comments, which significantly improved both the contents and the presentation. This work was supported in part by the US National Science Foundation under grants EIA–0080123, CCR–0113633, and CNS–0410918.

## REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.
[2] J. Anderson, "Computer Security Technology Planning Study," Technical Report ESD-TR-73-51, vol. II, US Air Force, Command and Management Systems, Bedford, Mass., Oct. 1972.
[3] M. Bernaschi, E. Gabrielli, and L. Mancini, "Operating System Enhancements to Prevent the Misuse of System Calls," *Proc. ACM Conf. Computer and Comm. Security,* pp. 174-183, 2000.
[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the Correctness of Memories," *Algorithmica,* vol. 12, nos. 2-3, pp. 225-244, Aug. 1994.
[5] M. Blum and S. Kannan, "Designing Programs that Check Their Work," *J. ACM,* vol. 42, no. 1, pp. 269-291, Jan. 1995.
[6] S. Chen, J. Xu, E.C. Sezer, P. Gauriar, and R. Iyer, "Non-Control-Data Attacks Are Realistic Threats," *Proc. USENIX Security Symp.,* Aug. 2005.
[7] U. Elingsson and F. Schneider, "SASI Enforcement of Security Policies: A Retrospective," *Proc. New Security Paradigms Workshop,* pp. 87-95, Sept. 1999.
[8] T. Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," *Proc. Network and Distributed Systems Security Symp.,* Feb. 2003.
[9] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A Delegating Architecture for Secure System Call Interposition," *Proc. Network and Distributed Systems Security Symp.,* Feb. 2004.
[10] J. Geovedi, J. Nazario, N. Provos, and D. Song, "Project Hairy Eyeball," http://blafasel.org/~floh/he/, 2005.
[11] J.T. Giffin, S. Jha, and B.P. Miller, "Detecting Manipulated Remote Call Streams," *Proc. 11th USENIX Security Symp.,* Aug. 2002.
[12] J.T. Giffin, S. Jha, and B.P. Miller, "Efficient Context-Sensitive Intrusion Detection," *Proc. Network and Distributed System Security Symp.,* Feb. 2004.
[13] B. Gladman AES Combined Encryption/Authentication Library, http://fp.gladman.plus.com/AES/index.htm, 2006.
[14] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, "A Secure Environment for Untrusted Helper Applications," *Proc. Sixth Usenix Security Symp.,* 1996.
[15] S. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *J. Computer Security,* vol. 6, no. 3, pp. 151-180, 1998.
[16] T. Iwata and K. Kurosawa OMAC: One-Key CBC MAC, 2002.
[17] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," *Proc. Network and Distributed Systems Security Symp.,* pp. 19-34, Feb. 2000.
[18] M. Jones, "Interposition Agents: Transparently Interposing User Code at the System Interface," *Proc. 14th ACM Symp. Operating Systems Principles (SOSP),* pp. 80-93, Dec. 1993.
[19] E. Krell and B. Krishnamurthy, "COLA: Customized Overlaying," *Proc. Winter 1992 Usenix Conf.,* pp. 3-7, Jan. 1992.
[20] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," *Proc. Eighth European Symp. Research in Computer Security (ESORICS '03),* pp. 326-343, 2003.
[21] C.M. Linn, M. Rajagopalan, S. Baker, C. Collberg, and J.H. Hartman, "Protecting against Unexpected System Calls," *Proc. Usenix Security Symp.,* pp. 239-254, Aug. 2005.
[22] M. Naor and K. Nissim, "Certificate Revocation and Certificate Update," *Proc. Seventh USENIX Security Symp.,* pp. 217-228, Jan. 1998.
[23] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Software," *ACM Trans. Programming Languages and Systems,* vol. 27, no. 3, pp. 477-526, May 2005.
[24] G. Necula and P. Lee, "Safe Kernel Extensions without Run-Time Checking," *Proc. Operating System Design and Implementation (OSDI),* pp. 229-243, Oct. 1996.
[25] N. Provos, "Improving Host Security with System Call Policies," *Proc. 12th USENIX Security Symp.,* pp. 257-272, Aug. 2003.
[26] R. Schlichting and F. Schneider, "Fail-Stop Processors: An Approach to Designing Fault Tolerant Computing Systems," *ACM Trans. Computer Systems,* vol. 1, no. 3, pp. 222-238, Aug. 1983.
[27] B. Schwarz, S. Debray, and G. Andrews, "Plto: A Link-Time Optimizer for the Intel IA-32 Architecture," *Proc. 2001 Workshop Binary Translation (WBT '01),* 2001.
[28] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," *Proc. IEEE Symp. Security and Privacy,* pp. 144-155, 2001.
[29] R. Sekar and P. Uppuluri, "Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications," *Proc. Eighth USENIX Security Symp.,* pp. 63-78, 1999.
[30] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney, "Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications," *Operating Systems Rev.,* vol. 37, no. 5, pp. 15-28, Dec. 2003.
[31] K.M.C. Tan and R.A. Maxion, ""Why 6? Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector," *Proc. 2002 IEEE Symp. Security and Privacy,* p. 188, 2002.
[32] V. Venkatakrishnan, R. Peri, and R. Sekar, "Empowering Mobile Code Using Expressive Security Policies," *Proc. 2002 Workshop New Security Paradigms,* pp. 61-68, 2002.
[33] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," *Proc. IEEE Symp. Security and Privacy,* pp. 156-169, 2001.
[34] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *Proc. IEEE Symp. Security and Privacy,* pp. 133-145, 1999.
[35] A. Wespi, M. Dacier, and H. Debar, "Intrusion Detection Using Variable-Length Audit Trail Patterns," *RAID '00: Proc. Third Int'l Workshop Recent Advances in Intrusion Detection,* pp. 110-129, 2000.

**Mohan Rajagopalan** received the BE degree from Mumbai University and the MS and PhD degrees from the University of Arizona. He is a research scientist in the Programming Systems Lab at Intel Corp. His research explores topics in both programming language and operating system design and implementation, with the goal of improving overall system characteristics such as performance, dependability, and security. He was a recipient of the 2005 IEEE/IFIP William C. Carter award for his work on Authenticated System Calls. He is a member of the IEEE.

**Matti A. Hiltunen** received the MS degree in computer science from the University of Helsinki and the PhD degree in computer science from the University of Arizona. He is a researcher in the Dependable Distributed Computing and Communication Department at AT&T Labs-Research in Florham Park, New Jersey. He is a member of the ACM and the IEEE Computer Society. His research interests include dependable distributed systems and networks, grid computing, and pervasive computing.

**Trevor Jim** received the BSE degree from Princeton University and the MS and PhD degrees from the Massachusetts Institute of Technology. He is a researcher in the Dependable Distributed Computing and Communication Department at AT&T Labs. He works in the areas of computer security and programming languages.

**Richard D. Schlichting** received the BA degree in mathematics and history from the College of William and Mary, and the MS and PhD degrees in computer science from Cornell University. He is currently director of software systems research at AT&T Labs-Research in Florham Park, New Jersey. He was on the faculty at the University of Arizona from 1982-2000, and spent sabbaticals in Japan in 1990 at Tokyo Institute of Technology and in 1996-97 at Hitachi Central Research Lab. Dr. Schlichting is an ACM Fellow and an IEEE Fellow, and is on the editorial board of the *IEEE Transactions on Software Engineering*. He is also the current chair of the IFIP Working Group 10.4 on dependable computing and fault tolerance, and has been active in the IEEE Computer Society Technical Committee on fault-tolerant computing, serving as its chair from 1998-1999. His research interests include distributed systems, highly dependable computing, and networks.