On Effective Model-Based Intrusion Detection

Jonathon T. Giffin Somesh Jha Barton P. Miller

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin

Technical Report 1543

Abstract

Model-based intrusion detectors restrict program execution to a previously computed model of expected behavior. We consider two classes of attacks against these systems: bypass attacks that evade detection by avoiding the detection system altogether, and transformational attacks that alter a detected attack into a semantically-equivalent attack that goes undetected. Recent detection approaches are problematic and do not effectively address these threats. We see reductions or outright failures in effectiveness and efficiency when systems (1) monitor execution at the library call interface, (2) provide accuracy via inlining of statically-constructed program models, or (3) use simplistic analysis of indirect function calls. Attacks can defeat library-call monitors by directly executing operating system kernel traps. Inlined models grow exponentially large at the trap interface: models for several test programs are 12,000 to 38,000 times larger at the trap interface than at the library call interface. Naïve indirect call analysis produces models 14 to 177 times larger than models built with in-depth analysis and that are less able to detect attacks. In examining these issues, our aim is to reveal complexities of model-based detection that have not been previously well understood.

1 Introduction

Host-based intrusion detection systems identify attempts to exploit program vulnerabilities, frequently by monitoring the program's execution. A model-based or behavioral-based anomaly detector [6] restricts execution to a precomputed model of expected behavior. An execution monitor verifies a stream of system calls generated by the executing program and rejects any call sequences deviating from the model. Constructing a model that balances the competing needs of detection ability and efficiency is a challenging task.

A successful attack subverts the execution of a vulnerable process in a manner undetectable to an execution monitor. We consider two threat models meeting this definition. Bypass attacks exploit design deficiencies of a detection system to avoid the execution monitor and generate arbitrary unmonitored system calls [2]. The system calls executed by the attack may not be allowed by the execution monitor; unfortunately, the monitor never intercepts the calls of a bypass attack. Transformational attacks, such as a mimicry attack [18, 27, 29], alter a detected attack so that it goes undetected by the model-based detection system yet carries the same malicious intent. A transformational attack is allowed by the program model.

Recently proposed model-based detection systems do not effectively address these threats. Resistance to bypass attacks requires enforcement of an interface that all attacks are required to use. This requires identification of the trusted computing base—the components of a computer system trusted as non-malicious and impenetrable to attack. In common host-based intrusion detection scenarios, the trusted computing base usually includes the operating system kernel and execution monitor, but no other code on the system. Effective system call interception requires monitoring of the kernel trap interface. Bypass attacks can defeat recent detection systems that monitor at other interfaces, such as the library call interface [16,17], by directly trapping to the kernel.

Resistance to transformational attacks requires program models to accurately represent correct execution behavior. Recent systems using static program analysis to build models [8,16,19,28] do not address the limitations of the analysis [20]. An accurate model of control flows at indirect function call sites requires identification of the possible targets of each indirect call. A simplistic indirect call analysis can offer opportunities to an attacker by including incorrect control flows in the model. Transformational attacks become easier to mount as the number of allowed execution paths and system call sequences increases. In our experiments on a UNIX system, the weak analysis used by a recent system [19] increased model size by 13 to 177 times and decreased model precision by 17 times when compared to models constructed with deeper indirect call analysis.

An accurate model must also correctly characterize function calls and returns. Model inlining is a recently proposed technique that addresses this need [16, 19]. Unfortunately, inlined models can not be efficiently enforced by an execution monitor. To be reasonably deployed, a monitor must rapidly verify system calls and maintain a low in-memory footprint. Inlined models grow exponentially with the height of a program's call graph and impose unreasonable requirements on verification. In our experiments, an inlined model was 71 times slower and required 83 times more memory than a non-inlined model of equal accuracy.

In examining these issues, our aim is to reveal attacker threats and model construction complexities that affect model-based intrusion detection systems. We believe that this paper makes the following contributions:

- Explicit enumeration of attacker threats that model-based anomaly detectors must address. Section 2 describes both the bypass attack and the transformational attack in further detail.

- An in-depth analysis of the interface at which an execution monitor can securely enforce a program model. We show in Section 3 that resistance to bypass attacks requires a non-circumventable interface. In most computer systems, the set of traps to the operating system kernel defines the entire secure interface. Attackers can trivially bypass recent systems that monitor execution at the library call interface.

- A comparison of two techniques that accurately model function call and return behavior—automata inlining [16,19] and the non-inlined Dyck model [14]. A previous publication [16] reported results of an unfair comparison between an inlined model at the library call interface and a Dyck model at the kernel trap interface. Our static analysis infrastructure can construct both inlined and Dyck models at both the library call and the kernel trap interfaces, allowing for valid comparisons. Section 4 shows that inlined models become prohibitively large to build and enforce when constructed at the non-circumventable kernel trap interface.

- Examination of the effect of indirect function calls on statically-constructed models and a call for foundational research in this area. In Section 5, we give real examples of how current automated identification of the targets of indirect calls significantly overapproximates the set of possible targets.

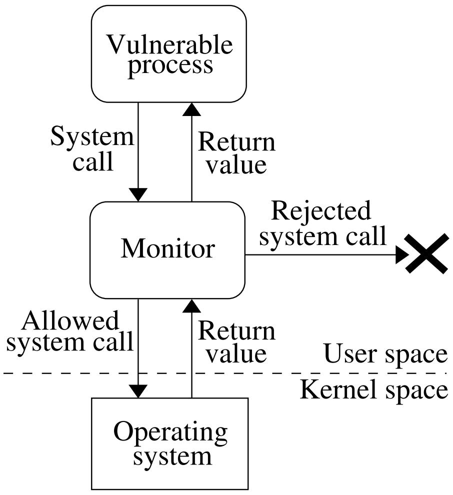

Figure 1: Monitored execution. The monitor intercepts system call requests and allows only calls matching the process' behavioral model.

2 Attacks Against Model-Based Anomaly Detectors

Host-based intrusion detectors find attacks by monitoring the behavior of locally executing processes. Model-based anomaly detectors verify a stream of system calls from an executing process against a pre-computed model of expected behavior [1, 3, 10, 21, 22, 26, 28, 30, 31].

An *execution monitor* only allows process execution that matches the program model. The process and its monitor execute above a trusted operating system (Figure 1), although the monitor may be incorporated into the operating system kernel for improved performance. To maintain its integrity, the monitor is isolated in a separate address space from the possibly subverted process that it observes. Note that such designs must be carefully architected so that new vulnerabilities, including time-of-check to time-of-use races between the monitor's verification of a system call and the subsequent kernel execution, are not introduced into the system [12]. Attackers attempt to alter a process' execution so that the process issues malicious system calls in a way undetectable to the monitor. Common attacks include execution of a command shell, appending of a new user to the system's password file, or privilege escalation.

The monitor can verify the process' execution at any programming interface with observable events, such as the kernel trap interface. Whenever the process calls the interface, the monitor intercepts the call and suspends execution of the process. If the call is allowed by the program model, the monitor resumes the process' execution. A call falling outside the program model indicates to the monitor that the process has been subverted, and any appropriate response can then be activated. The monitor thus protects the operating system from subverted processes.

The operating system and monitor form the *trusted computing base* (TCB) of the computer system. For any assurances of system security to be valid, we require the TCB to be non-malicious and free from exploitable vulnerabilities. Processes change the TCB only through a well-defined system call interface. All processes outside the TCB are explicitly untrusted, and we expect that an attacker can manipulate these processes in an arbitrary manner. Model-based intrusion detectors prevent manipulation from harming the TCB by identifying a process' attempts to use system calls in an unexpected way.

Successful detection of an attack requires two assumptions to hold. First, attacks must produce events observable to the monitor. Second, the model of expected behavior must be precise and not accept attacks as valid system call sequences. Attackers can evade detection with any attack that violates one or both assumptions. We consider both possibilities.

Bypass attacks evade detection by bypassing the monitored interface. These attacks exploit systems that monitor high-level interfaces and fail to enforce execution at the interface of the trusted computing base. By calling the trusted computing base directly rather than through a higher interface, the attack can maliciously alter the system without detection. We examine bypass attacks in Section 3.

Transformational attacks alter an existing attack A detected by a model-based anomaly detector into a semantically-equivalent attack A' that evades detection by appearing as valid execution [18, 27, 29]. The sequence of system calls in A' exists as a sentence in the language accepted by the program model. If the program could generate A' when executing correctly, then the model is correct and alternative detection approaches may be necessary. If the program could never generate A' in correct execution, then the program model is imprecise and new model construction strategies must be developed. We examine two techniques to build more precise program models, context-sensitive construction and indirect call analysis, in Sections 4 and 5.

3 Monitored Interface

Bypass attacks exploit design errors of an execution monitor to avoid the monitor and execute unverified system calls. To resist bypass attacks, a model-based intrusion detection system must verify a process' execution at the non-circumventable interface of the trusted computing base (TCB). This section addresses the following points:

- For most common operating systems, including Windows, Linux, and UNIX, the TCB usually includes only the operating system kernel and execution monitor. Code in shared object files and dynamically linked libraries (DLLs) are explicitly not part of the TCB.
- Bypass attacks can evade a model-based intrusion detection system that does not monitor the interface of the trusted computing base.
- Although current Windows exploits commonly call library functions in Windows subsystem DLLs, these exploits could be modified to directly execute Windows kernel traps.

Recent research has focused on execution monitoring at two different interfaces—library calls and operating system kernel traps. The library call interface seems reasonable: most programs make calls to standard library code, and library interpositioning is frequently an easy way to generate events during execution. However, library code is not part of the trusted computing base. We will show that this interface is circumventable and cannot provide security for secure systems. We subsequently consider sequences of kernel traps generated by programs and claim that this is the only non-circumventable interface useful to execution monitors.

The implications of choosing an improper interface are profound. Efforts to enforce correct patterns of use on any circumventable interface will not be secure. Bypass attacks can escape such monitoring by simply bypassing the interface. Furthermore, we will show in Section 4 that poorly choosing the event interface compounds the problems of algorithm design for accurate model construction.

3.1 Library Call Interface

Nearly every process running on a modern operating system executes code both from the application and from shared libraries. We make a distinction between application code and library code. *Application code*

```
void main (int, char **argv) {
  syslog(0, argv[0]);  // Format string vulnerability
  setuid(0);           // Need root uid for chroot
  chdir("/var/www");   // Need chdir before chroot
  chroot(".");         // Establish chroot jail
  if (scandir(".", NULL, isSetuid, alphasort) > 0)
    exit(1);           // Exit if jail contains setuid program
  setuid(101);         // Drop privilege from root
  exec(argv[1]);       // Exec any program inside jail
}
```

Figure 2: Code example. The function `isSetuid` is a user-provided function that takes a directory entry and returns true when that entry is a setuid binary.

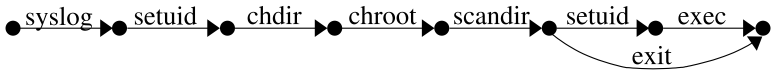

Figure 3: Library model.

includes the binary code sections contained in the executable image file loaded by the operating system. This code is generally unique to the program and was written for a specific use. *Library code* is binary code in shared object files, including the runtime linker, the standard C library, and Windows subsystem DLLs. This is general-purpose code used by many processes and does not reside in the image file of the application.

In many ways, some functions in the standard libraries can be viewed as a user-space extension of the kernel. These library calls serve two basic purposes:

1. **Convenience.** A process makes an operating system request by executing a software trap. The trap requires a particular machine instruction, such as `int`, `systrap`, or `syscall`, that is generated from hand-written assembly code rather than from a high-level language compiler. Shared libraries can provide convenient wrapper functions around kernel traps, providing applications with a function call interface to the kernel.

2. **Error checking.** The library wrapper functions examine error returns from kernel traps and provide a higher-level error interface to application programs.

Hence, we can reasonably expect programs to make use of shared libraries because it is easy and convenient to do so. As a direct consequence, we can expect any kernel trap executed by a process to have been preceded by some call to a library function.

Recent model-based intrusion detection systems make use of this intuition and monitor execution at the library call interface [16, 17]. This interface offers advantages: the library call interface is much richer and can better fingerprint correct process execution. For example, Windows, UNIX, and Linux kernels each have fewer than 256 traps, but the Windows system library has nearly 1200 entry points. The Solaris C library has over 2000 addressable functions.

The rich library interface can build expressive execution models. Consider the example function `main` in Figure 2 that makes a series of library calls. After logging its filename, it creates a chroot jail in `/var/www`, ensures that the jail directory contains no setuid executable, drops privilege, and executes any program in the jail. The `syslog` call contains a format string vulnerability that allows an attacker to write to arbitrary memory locations, possibly changing the program's execution following `syslog` to jump directly to the

```
/* setuid(0) */
"\x31\xc0" // xorl %eax,%eax
"\x31\xdb" // xorl %ebx,%ebx
"\xb0\x17" // movb $0x17,%al
"\xcd\x80" // int $0x80
```

Figure 4: A fragment of shellcode used by a format string attack. The final four bytes directly execute the `setuid` kernel trap. The preceding four bytes construct the arguments passed to the kernel trap handler.

`exec` call. Systems that enforce correct use of library calls analyze this code to build the automaton model of library call events shown in Figure 3.

Unfortunately, *these systems offer no real security*. The security of systems monitoring execution at the library call interface requires that the only way to execute a kernel trap is by first calling a library function. Although library code may appear to be an extension of the kernel, this is a fallacy. Libraries are not part of the TCB, and the library interface is *circumventable*, allowing attack code to execute kernel traps without first calling a library function. The attacker exploiting the format string vulnerability in the example code can transfer execution into machine instructions contained in their format string that execute kernel traps directly (Figure 4). This bypass attack invokes kernel operations but escapes any process monitor intercepting library calls.

3.2 Kernel Trap Interface

However, the attack cannot bypass the kernel. As part of the trusted computing base, the kernel code is immune from direct attack and can only be entered via known entry points. These entry points define a secure interface that attacks cannot circumvent. An attack that generates malicious system calls must create kernel trap events. An execution monitor verifying a process' execution at the kernel trap interface can intercept the trap and detect the malicious behavior. An attack can evade the monitor by failing to execute kernel traps, but any such attack would be contained to the process.

Regarding their work on Linux, Jones and Lin write:

> The library call approach works well with buffer overflow attacks when, *as is typical*, the attacker code adds new sequences of library calls . . . [17] (emphasis added)

We disagree. In our experience, typical Linux and UNIX attacks do not call library functions. On UNIX-like systems, it is often easier for attacks to directly execute kernel traps rather than call library functions. The kernel trap interface is well known and easy to invoke. Trap use requires no knowledge of the address at which the library code resides in memory. These observations hold in practice: We surveyed Linux and UNIX exploits archived by SecuriTeam between 1 January 2004 and 2 May 2005 [25]. Of 47 code injection attacks, including buffer overflows and format string attacks, 46 injected code that bypassed the library interface and directly executed kernel traps. For example, Figure 4 shows a fragment of code injected and executed in a format string attack against a web server front-end load balancer [5]. Intrusion detection systems monitoring the library call interface would not even detect these current attacks.

Current exploits against processes executing on Windows work differently. All operating system kernel traps are intended to be executed only by Windows subsystem shared libraries and not directly by applications. Windows exploits largely obey this programming practice, so library call enforcement would detect most current Windows attacks.

This exploit design is largely an artifact arising from the obfuscated set of Windows kernel traps. The kernel trap interface is not widely published and may change between operating system releases. Yet, details of the Windows trap interface, called the "Native API", are available to interested attackers [23]. These attackers can convert an attack that calls subsystem library functions into an attack that directly invokes Windows kernel traps. Although library call verification may detect today's attacks, with little effort attackers can alter their Windows exploits to bypass the library call interface.

Recent intrusion detection systems monitor a combination of kernel traps and function call return addresses stored on the call stack [8, 9, 14]. It is important to understand the utility of function call monitoring given the knowledge that the function call interface is circumventable [11, 18].

Function call and return events serve only to improve the efficiency of online execution monitoring. Only the kernel traps provide a secure monitoring point. Our system and the system of Feng *et al*. build automata accepting the context-free language of kernel traps that a correctly executing process can generate. Recognizing a context-free language incrementally is a cubic-time operation and too slow for real-world deployment [30]. However, monitoring function calls determinizes automaton operation and allows the systems to recognize a context-free language in linear time. The function call events offer no additional security but significantly improve the efficiency of kernel trap verification. Attacks can produce fake function call events but are still limited to the context-free language of kernel traps accepted by the model.

Gopalakrishna *et al*. recognized that attacks can trivially escape library call monitoring, and suggested that a combination of library call and kernel trap enforcement may be useful [16]. Such enforcement is subsumed by systems that monitor both traps and the entire set of function calls, as library calls appear as just another function call to these systems.

3.3 Changing the Trusted Computing Base

Secure execution monitoring occurs at the interface to the trusted computing base. If the TCB changes, then the monitored interface must likewise shift. Remote and distributed execution environments reflect such a change. Processes execute on remote, untrusted machines and communicate with the parent process of the distributed computation via a stream of events. The entire remote machine, including the operating system and the hardware, may be malicious [13, 24]. The trusted computing base includes the machine on which the parent process executes and the parent process itself.

An attacker at the remote machine may attempt to harm the parent process by sending malicious events to the parent. This communication channel is the non-circumventable interface that an attacker is unable to bypass. A model-based anomaly detection system can first model the remote process' use of the communication channel, and subsequently monitor the channel to detect subversion of the remote process as deviation from the model.

4 Constructing Efficient Context-Sensitive Models

To reduce an attacker's opportunity to construct transformational attacks, a program's model must accurately characterize the program's possible control flows. One construction method used recently to accurately model function calls and returns inlines models of called functions [16, 19]. We show that techniques that replicate model structure, such as inlining, build prohibitively large models at the secure kernel trap interface:

- In our experiments on a UNIX system, we could not construct inlined models at the kernel trap interface for two test programs, `htzipd` and `gnatsd`, because the construction consumed the entire 4 GB virtual address space of our machine.

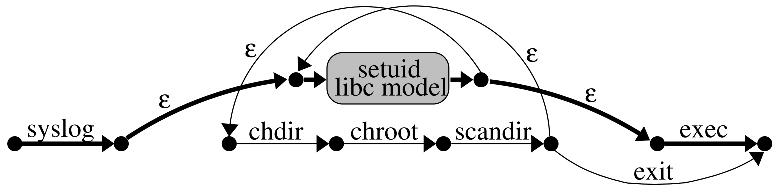

Figure 5: Call-site replacement applied to the C library call `setuid`. The simplistic algorithm used introduces an impossible path, shown in bold, because it does not correctly model function calls and returns.

- We successfully constructed inlined models for two other programs, `cat` and `lhttpd`. The inlined models were 360 to 400 times larger than corresponding non-inlined models.
- Choosing a poor interface compounds the difficulty of finding reasonable model construction algorithms. Although of reasonable size at the bypassable library interface, our inlined models became 12,000 to 38,000 times larger at the secure kernel trap interface.
- The large size of inlined models leads to prohibitive runtime enforcement resource demands. Inlined models were 71 times slower and required 83 times more memory than models of equal accuracy that used different construction techniques.

Static analyzers construct a program model in two stages:

1. First analyze each function, abstracting away all code except for control-flow transfers like branches, function calls, and kernel traps.
2. Second, assemble the models for each function into a global model for the program. The designer of the system must choose an algorithm for *call-site replacement* that dictates how to model interprocedural control flows like function calls and returns.

Good call-site replacement strategies must keep models compact, accurately characterize interprocedural control flows, and produce models efficient to operate during online execution verification. These strategies are not trivial to develop and are the topic of this section.

Figure 5 shows the model for `main` of Figure 3 with a simplistic call-site replacement strategy applied to the library call `setuid`. The original call edges targeting `setuid` have been replaced with ϵ-edges transferring control into and out of the model for `setuid`. This strategy is context insensitive and does not enforce proper function call and return behavior. Context insensitive models are subject to transformational attacks, such as impossible path attacks [28] and mimicry attacks [18, 27, 29], that exploit this inaccuracy. The bold path in Figure 5 is an impossible path accepting altered program executions that bypass chroot jail creation and loss of root privilege.

4.1 Push-Down Automaton

A push-down automaton (PDA) accepts a context-free language such as that generated by a program executing with proper call and return semantics. The call-site replacement algorithm constructs a PDA by adding push and pop symbols to the ϵ-edges of Figure 5. Each call site uses distinct symbols corresponding to the distinct function call return addresses used in the program. The execution monitor rejects any event sequences not contained in the context-free language accepted by the PDA. This algorithm keeps model size compact, as all calls to a particular function link to the same model. There is no replication of state.

Program	*Application only*	*With libraries*
cat	1,232	185,844
htzipd	16,485	337,560
lhttpd	1,826	313,014
gnatsd	33,853	351,279
sendmail	133,915	688,387

Table 1: SPARC instruction counts.

Feng *et al*. [8, 9] and our previous work [14] used function call and return events to keep the cost of online PDA operation to time linear in the PDA size.

4.2 Inlining

The additional call and return events do impose a slight cost, as the execution monitor must read data from the virtual address of the monitored process at each system call. Two recent publications suggested an alternative method to retain much of a model's context sensitivity while dispensing with the need to compute call and return events. Lam and Chiueh [19] and Gopalakrishna *et al*. [16] implemented call-site replacement as an algorithm that inlined models at call sites.

Inlining replaces a function call transition in an automaton with a unique copy of the target function's model. Even though a program may contain multiple call sites all targeting the same function, this algorithm does not introduce impossible paths because each call site links to a different copy of the target model. Inlining cannot be used at recursive call sites, so the final program model accepts a regular language over-approximation of the context-free language of events that the executing process can generate. A regular language can be recognized in linear time without requiring the additional function call and return events.

However, inlining replicates models of functions, raising concerns that it will not scale to large programs. Gopalakrishna *et al*. presented reasonable results for three of four test programs when building their inlined IAM model for the insecure library interface. Unfortunately, this poor interface selection obscures inlining's performance when constructing models at the non-circumventable kernel trap interface. Intuition suggests that the space cost of inlining is exponential in the height of the call graph. Consider a simple example: if function f is a leaf function called by four other functions, and each of those four other functions are called by five other functions, the number of copies of the model for f is already $4 \cdot 5$. This multiplication repeats for the entire height of the call graph and for every leaf function in the program.

Modeling a program at the kernel trap interface increases the number of functions that must be modeled and the height of the call graph by including code from shared libraries. The first four programs listed in Table 1 correspond to the test programs used by Gopalakrishna *et al*. in their previously published results. Using static binary analysis of SPARC executables, we constructed inlined models for these programs at both the library interface and the kernel trap interface to measure the ability of inlining to scale to complete programs. From our earlier intuition, we expected inlined models constructed for the trap interface to be significantly larger than models for the library interface.

Tables 2–5 show the results of our comparisons of the static analysis demands for the four test programs used by Gopalakrishna *et al*. The static analyzer executed on a Sun Microsystems SunFire V880 server with dual 32-bit 750 MHz UltraSparc III processors, 4 GB of physical memory, and running Solaris 8. Solaris allows user processes to address the entire 4 GB memory space. We list the memory used by the static analyzer, the time required to construct the program model, and the size of the resulting model for

Model	Interface Library calls	Kernel traps
Binary IAM	5.4 MB	760 MB
Dyck	5.4 MB	8 MB

(a) Model construction memory use.

Model	Interface Library calls	Kernel traps
Binary IAM	0.48 sec	515 sec
Dyck	0.39 sec	4 sec

(b) Model construction time.

Model	Interface Library calls	Kernel traps
IAM [16]	90 states 791 edges	Unreported
Binary IAM	53 states 123 edges	17,338 states 4,689,814 edges
Dyck	57 states 117 edges	1,737 states 12,882 edges

(c) Automaton sizes.

Table 2: Results for `cat`.

Model	Interface Library calls	Kernel traps
Binary IAM	19 MB	$>$ 4082 MB
Dyck	8.4 MB	16 MB

(a) Model construction memory use.

Model	Interface Library calls	Kernel traps
Binary IAM	16 sec	$>$ 1382 sec
Dyck	2 sec	20 sec

(b) Model construction time.

Model	Interface Library calls	Kernel traps
IAM [16]	2,821 states 31,047 edges	Unreported
Binary IAM	2,943 states 20,796 edges	Out of memory
Dyck	593 states 1,377 edges	2,167 states 14,812 edges

(c) Automaton sizes.

Table 3: Results for `htzipd`.

each test program. We list values both for model construction using the circumventable library interface and for construction using the secure kernel trap interface. Results for the IAM model are not computed but are copied from the publication of Gopalakrishna *et al*. [16]. That publication did not present results for memory use, build time, or models constructed at the trap interface, so we implemented inlining as a call-site replacement technique in our existing binary analysis infrastructure. We computed results shown as Binary IAM using this infrastructure. We lastly include results for construction of our Dyck model [14], which uses the PDA call-site replacement algorithm.

The Binary IAM models at the library interface are significantly smaller than the IAM models constructed by Gopalakrishna *et al*. from source code analysis. We examined the inlined model for `cat` and identified two analysis differences that accounted for the much of the discrepancy. First, our infrastructure performed more aggressive optimization of program models than did the infrastructure of Gopalakrishna *et al*. In particular, we minimized the automata to remove redundant edges. Second, the source code analyzed by Gopalakrishna *et al*. did not match object code produced by a compiler. The source code was a strange mix of preprocessed and non-preprocessed code that had some macros expanded and some remaining [15]. As a result, the IAM models contained macros as events even though library interpositioning could never intercept a macro event. By disabling optimizations and manually inserting macro events, we could increase the size of the Binary IAM model for `cat` to 100 states and 640 edges. Manual verification indicated that our analyzer was executing correctly and that our Binary IAM models are better optimized than the source-level IAM models previously reported.

Model	Interface Library calls	Kernel traps
Binary IAM	6.5 MB	2328 MB
Dyck	5.7 MB	14 MB

(a) Model construction memory use.

Model	Interface Library calls	Kernel traps
Binary IAM	3 sec	919 sec
Dyck	0.47 sec	20 sec

(b) Model construction time.

Model	Interface Library calls	Kernel traps
IAM [16]	429 states 1,098 edges	Unreported
Binary IAM	280 states 459 edges	21,445 states 5,567,470 edges
Dyck	212 states 455 edges	2,050 states 13,774 edges

(c) Automaton sizes.

Table 4: Results for `lhttpd`.

Model	Interface Library calls	Kernel traps
Binary IAM	1582 MB	> 4083 MB
Dyck	15 MB	45 MB

(a) Model construction memory use.

Model	Interface Library calls	Kernel traps
Binary IAM	870 sec	> 1588 sec
Dyck	25 sec	64 sec

(b) Model construction time.

Model	Interface Library calls	Kernel traps
IAM [16]	338,736 states 7,915,678 edges	Unreported
Binary IAM	1,600,999 states 7,888,767 edges	Out of memory
Dyck	2,105 states 8,928 edges	6,827 states 284,715 edges

(c) Automaton sizes.

Table 5: Results for `gnatsd`.

The tables contain important features:

- Inlined models grow prohibitively large for even small programs when modeling the kernel trap interface. The model for `cat` grew by *3.8 million percent*. Attempts to construct models for `htzipd` and `gnatsd` failed after consuming the entire 4 GB address space of the 32-bit system.
- The non-inlined Dyck model better scales with increasing code size and code complexity.
- Even when building models at the library interface, the non-inlined Dyck models are smaller than their inlined counterparts.

We also measured the ability of inlined models to enforce execution at the kernel trap interface. We executed `cat` with the command-line option "-n" and a workload that wrote 38 files totaling 500 MB to disk. Both the Binary IAM model and the Dyck model effectively constrained an attacker. Measurements of precision were identical to three decimal places, indicating that *the inlined model's loss of precision at recursive call sites was not significant.*

Inlined models faired less well when evaluating their performance impact. Table 6 shows the slowdown in runtime due to execution verification. Both the Binary IAM and Dyck models can be operated in time linear in the automaton size. The significant increase in the inlined model's size thus resulted in significantly worse performance. Table 7 gives the memory use demands of our runtime monitor for the two models. The large inlined model required significant system resources.

The architecture of the Windows operating system will only worsen the effects of inlining. Given its microkernel design, the user-space subsystem DLLs contain significant functionality that models constructed for the library call interface will not include. However, shifting execution monitoring to the secure Windows

Model	Unverified execution	Verified execution				
		Initialization	Parse model	Execution	Total	Slowdown
Binary IAM	55.32 s	0.06 s	6.75 s	166.57 s	173.38 s	214 %
Dyck	55.32 s	0.02 s	0.06 s	56.90 s	56.98 s	3 %

Table 6: Performance impact of verifying `cat`'s execution against a program model.

Model	`cat` memory use	Model size	Increase
Binary IAM	976 KB	19,936 KB	2043 %
Dyck	976 KB	240 KB	25 %

Table 7: Memory use impact of execution monitoring.

kernel trap interface requires any execution model of a Windows application to additionally model code from the libraries. Consider an example: the simple editor `notepad.exe` contains 85 functions. The DLLs required by `notepad` together contain 22,120 functions.

These results speak emphatically about the inability of inlined models to meet the needs of practical security systems. We believe that Gopalakrishna *et al.* were able to realize relatively small inlined models for `cat`, `htzipd`, and `lhttpd` because the application code size of these three programs was small enough that inlining had not yet reached the rapid growth of the exponential growth curve. As they reached `gnatsd`, explosive growth becomes evident even at the library interface. Adding library code to the programs pushes all models into rapid growth.

Lam and Chiueh constructed inlined models at the Linux kernel trap interface without the exponential model growth presented here [19]. Their results are possible for three reasons. First, Linux standard library code is less complex than the Solaris standard libraries used in our experiments. Second, Lam and Chiueh did not statically inline models at indirect call sites and instead used a monitoring-time call-site replacement algorithm that did not replicate model state. Third, they analyzed statically-linked programs that contained no runtime linker. In typical dynamically-linked programs, every library call may invoke the runtime linker due to lazy linking and lazy loading. The runtime linker is complex and significantly increases model complexity when used.

5 Effects of Indirect Calls

Indirect function calls add control-flow complexity that may require deep analysis to satisfactorily resolve. Indirect calls significantly affect program models constructed from static analysis:

- Omission of indirect call analysis builds models 7 to 144 times larger than models built with analysis and 14 to 177 times larger than models built with a combination of analysis and manual annotation.
- Indirect call analysis can significantly constrain attackers. Omitted or weak analyses increase an attacker's opportunity to develop a transformational attack.
- Current automated data-flow analyses are insufficient and require manual annotation.

Static code analyzers must identify all possible targets of indirect function calls so that the generated model correctly characterizes all control-flow transfers. For example, the library function `scandir`, called in Figure 2, takes two function pointers as arguments and subsequently calls them via indirect call sites. Analysis

Program	*Indirect call analysis*		
	None	*Data-flow analysis*	*Annotations*
cat	5,179 states 2,285,988 edges	1,643 states 15,849 edges	1,737 states 12,882 edges
htzipd	5,748 states 2,303,075 edges	2,082 states 17,958 edges	2,167 states 14,812 edges
lhttpd	5,378 states 2,286,568 edges	1,977 states 17,499 edges	2,050 states 13,774 edges
gnatsd	13,566 states 8,828,375 edges	5,908 states 455,809 edges	6,827 states 284,715 edges
sendmail	27,286 states 15,613,707 edges	18,591 states 2,118,193 edges	18,478 states 1,149,040 edges

Table 8: Effect of pointer analysis on Dyck model size.

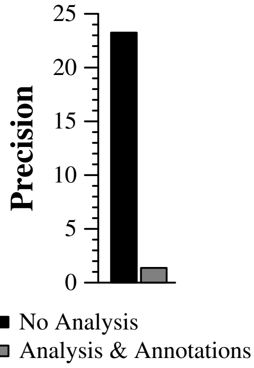

Figure 6: Model precision for `cat` with indirect call analysis. A lower bar is better.

of this code must identify the targets of these indirect calls. This identification is undecidable for both source and binary code analysis [20], so statically-constructed models can only approximate the set of possible targets. The most naïve approach, used by Lam and Chiueh [19], performs no analysis and allows an indirect function call to target any function in the program. This method greatly overapproximates correct program behavior and gives an attacker significant freedom. Wagner and Dean constrained the set of targets to only those functions whose address is taken in the program's source code [28]. Again, this significantly overapproximates correct execution by treating all function pointers as identical. Gopalakrishna *et al*. improved this analysis by separately analyzing each function pointer in C source code [16]. They allowed an indirect function call to target only those functions whose type signature matched that of the call site.

Binary code analysis is of greater difficulty because binary code is weakly typed. Our previous work used binary data-flow analysis to compute the addresses used at indirect function calls [13]. When data-flow analysis could not recover a call site's possible targets, the constructed model allowed the site to target any function with its address taken, as computed from data-flow analysis. Significantly, implementation limitations caused the analyzer to miss certain targets and introduce false alarms. For example, the analyzer could not recover function addresses in C++ vtables when those vtables were used by objects on the heap.

Manual annotation can improve the precision of these analyses. Wagner and Dean manually restricted the possible targets of some indirect function call sites to improve the precision of their models. We used annotation to similarly improve precision and to add control flows missed by the automated data-flow analysis. Annotations can be relatively simple: the pair ("load_so", "elf_map_so") specifies that the indirect call in "load_so" targets the function "elf_map_so"; or inconvenient: the annotation (0x1adf8, "__db_set_lorder") includes the virtual address of an indirect call site in the object code of a library. Virtual addresses must be used to distinguish among multiple indirect call sites in one function.

These analyses significantly alter the constructed models. Table 8 shows the sizes of the PDA-based Dyck model constructed with varying levels of indirect call analysis. We show results for the four programs used in earlier sections and for `sendmail`, a program with 688,387 SPARC instructions in its application code and library code. The table includes model sizes with Lam and Chiueh's omitted pointer analysis, our automated binary data-flow analysis, and with manual annotation. Figure 6 presents model precision measurements, using average branching factor [28], for executions of `cat`. Lower measurements indicate less opportunity for attack and hence greater precision.

In each case, increasingly complex analyses improve model precision. Models constructed with indirect call analysis accept fewer event sequences and better constrain program behavior and attacker opportunities. Including the manual annotation step improves existing automated analyses by about 50% for some programs. Unfortunately, annotation is inconvenient and diminishes the usefulness of the work. We see a need for additional research to develop more complex binary data-flow analyses than those used in our previous work and note that several researchers have already begun making progress in this area [4, 7].

6 Conclusions

Model-based intrusion detection system design should consider the threats that the systems must face. Bypass attacks escape a monitor that verifies execution at a circumventable interface. Recent proposals to monitor executing processes at the library call interface are unable to address bypass attacks and can be successfully defeated by an attacker. The interface of the trusted computing base—frequently the operating system kernel trap interface—is the only secure monitoring point.

Transformational attacks modify a detected attack so that it appears to the monitor as correct execution. The models enforced by the monitor must accurately represent correct program behavior to reduce an attacker's opportunities to successfully develop a transformational attack. Function calls and returns can be correctly modeled with a push-down automaton and efficiently operated with call and return events in addition to kernel trap events. Approaches using automaton inlining see exponential growth of the program models as program size increases. Unfortunately, reasonable model size at the bypassable library call interface was not an indicator of reasonable models at the secure kernel trap interface.

Static model construction efforts remain ongoing, as evidenced by efforts to improve indirect function call analysis. As with function calls and returns, models must accurately characterize the possible control flows at indirect function call sites. Current automated analyses are insufficient and frequently require human annotation of indirect call targets to build precise models that produce no false alarms. We expect that continued research in static data-flow analysis will make solely automated algorithms effective.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, California, Feb. 2003.

[2] Anonymous, J. Butler, and Anonymous. Bypassing 3rd party windows buffer overflow protection. *Phrack*, 11(62), July 2004.

[3] K. Ashcraft and D. R. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *13th International Conference on Compiler Construction (CC)*, Barcelona, Spain, Apr. 2004.

[5] N. De. Pound format string exploit. http://www.securiteam.com/exploits/5RP052KCUI.html, May 2004.

[6] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31:805–822, 1999.

[7] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 1998.

[8] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[9] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[10] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2004.

[11] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[12] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2003.

[13] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.

[14] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *11th Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.

[15] R. Gopalakrishna. Private communication, Apr. 2005.

[16] R. Gopalakrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2005.

[17] A. K. Jones and Y. Lin. Application intrusion detection using language library calls. In *17th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, Louisiana, Dec. 2001.

[18] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *14th USENIX Security Symposium*, Baltimore, Maryland, Aug. 2005.

[19] L.-c. Lam and T.-c. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sophia Antipolis, France, Sept. 2004.

[20] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, Dec. 1992.

[21] W. Lee, S. J. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[22] N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, pages 257–272, Washington, DC, Aug. 2003.

[23] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. Microsoft Press, 4th edition, Dec. 2004.

[24] T. Sander and C. Tschudin. Protecting mobile agents against malicious hosts. volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, 1998.

[25] Securiteam.com exploits. http://www.securiteam.com/exploits/.

[26] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[27] K. M. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID) 2002, LNCS #2516*, pages 54–73, Zurich, Switzerland, October 2002. Springer-Verlag.

[28] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[29] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Nov. 2002.

[30] D. A. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD dissertation, University of California at Berkeley, Fall 2000.

[31] H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sophia Antipolis, France, Sept. 2004.