Tailored Application-specific System Call Tables

Qiang Zeng, Zhi Xin, Dinghao Wu, Peng Liu, and Bing Mao

ABSTRACT

The system call interface defines the services an operating system kernel provides to user space programs. An operating system usually provides a uniform system call interface to all user programs, while in practice no programs utilize the whole set of the system calls. Existing system call based sandboxing and intrusion detection systems focus on confining program behavior using sophisticated finite state or pushdown automaton models. However, these automata generally incur high false positives when modeling program behavior such as signal handling and multithreading, and the runtime overhead is usually significantly high. We propose to use a stateless model, a whitelist of system calls *needed* by the target program. Due to the simplicity we are able to construct the model via static analysis on the program's binary with much higher precision that incurs few false positives. We argue that this model is not "trivial" as stated by Wagner and Dean. We have validated this hypothesis on a set of common benign benchmark programs against a set of real-world shellcode, and shown that this simple model is, instead, very effective in preventing exploits. The model, encoded as an per-process tailored system call table, incurs virtually no runtime overhead, and should be practical to be deployed to enhance application and system security.

1. INTRODUCTION

System calls represent an important abstraction layer of services provided by an operating system kernel. A commodity operating system usually provides the same set of system calls to all the programs. In practice, however, no programs utilize all the services provided by the kernel. While a lot of system calls may never be used by a benign program, they can be exploited by the injected malicious code. Current operating systems provide very limited mechanisms to prevent system calls from being abused. For example, root privileges may be required for a few system services, and policies may be relaxed for *setuid* programs.[1] Once a program is compromised, a uniform set of kernel services are potentially exposed to attackers as a versatile and handy toolkit. While a uniform system call interface simplifies operating system design, it does not meet the principle of least privileges in the perspective of security, as the interface is not tailored to the needs of applications.

Existing researches on system call based sandboxing and intrusion detection focus on system call pattern checking [16, 12, 31, 35, 7], arguments validation [35, 23, 25, 4, 3], and call stack integrity verification [10, 15]. Some establish sophisticated models such as Finite State Automata (FSA) [31, 35, 15, 13] and Pushdown Automata (PDA) [35, 14] by training [12, 31, 34, 10] or static analysis [35, 14, 15, 13], while others rely on manual configuration to specify the rules and policies [16, 3, 7].

There are several severe *practicality* limitations in the existing approaches. First, the effort and cost of policy construction is usually high when manual configuration is involved. For example, from the point of view of users who intend to confine system call invocation of applications, how to efficiently and precisely construct system call policies without high false positives (denial of services)? Systrace [28], for instance, allows one to build system policies interactively. However, system calls usually represent low-level services provided by an operating system kernel, such that users typically have difficulty in identifying system call invocation rules in terms of types, sequences and arguments. Even program developers may find it obscure since system calls are generally not invoked directly but through library APIs. If no manual configuration is involved, from the intrusion detection point of view, how to identify or learn the intrusion pattern based on system call behaviors with low false positive rates? This leads to the second limitation below.

Second, it is difficult to construct a model that incurs few false positives without dramatically degrading detection effect. If a policy or model is constructed by training, it is hard to ensure every execution path has been exercised, while many theoretical and practical barriers remain for static analysis based approaches. In addition, it usually imposes a significant overhead to enforce the models and policies. Due to those limitations and unsolved questions, few sys-

[1]Due to this reason, they are the most common victims of privilege escalation attacks.

tem call based sandboxing and intrusion detection systems are deployed in practice.

In this paper, we propose System Interface Tailor (SIT) using a application-specific whitelist, instead of system call sequences, FSA, or other sophisticated models, to confine system calls used by a specific program. Given a program, its whitelist includes all the system calls needed by the program; any invocation to system calls not included in the whitelist are sandboxed or identified as intrusion at runtime. *Wagner and Dean [35] stated that the whitelist is a "trivial" model. We argue that this model is not only more feasible to be constructed automatically(, compared to constructing an automaton model), but can block a significant number of exploits. Moreover, it incurs virtually zero runtime overhead to enforce the model by representing it as a tailored application-specific system call table.*

Our hypothesis is that many programs use only a relatively small set of system calls, and intrusion such as code injection may invoke system calls that are not used by the original programs. We have validated this hypothesis on a set of popular server program, including *apache*, *bind*, and *sendmail*, and common benchmarks, against a set of real-world shellcode. The experiment results show that 35% of the shellcode attacks can be prevented from over 90% of the benign programs after applying our technique, while 80% of the benign programs can prevent 70% of the exploits.

Consider the system call execve as an example, about half of shellcode instances invoke this system calls, while only 15% of the benign programs use it. This means that, by merely disabling this system call, 85% of the benign programs can block about half of the shellcode exploits.

The whitelist is extracted from executables automatically by a static analyzer. Compared to models based on training or configuration, which is usually time-consuming and error-prone, our model construction is automatic. Due to the simplicity of the model, many limitations that exist in static analysis for FSA or PDA models can be overcome.

A whitelist can be precisely represented as a *tailored application-specific system call table* during runtime, which routes the valid system calls to original system service routines and the invalid ones to an intrusion response procedure. Each application has its own (virtual) system call table in the kernel, as opposed to a uniform system call table for all applications. The runtime enforcement overhead is near zero. Considering its extremely low overhead, the sandbox can be applied at system level to protect all the user programs and improve system security, which is another sharp contrast with other costly techniques that are not affordable by the system to protect many processes.

We have implemented our approach in the Linux platform. Our system consists of two parts: a static analysis module that can automatically extract from an executable all the system calls used, and a runtime enforcement module that enforces the whitelist blocking all the system calls not included in this list.

Our stateless model avoids a lot of challenges faced by stateful automaton models, such as signal handling and multi-threading; it also simplifies the analysis of indirect calls by exploiting the information contained in the binary. So it incurs few false positives. We also ran our benchmark programs with varying inputs and did not encounter denial of service. Our efficiency test has shown that the per system call invocation overhead is less then 7 CPU cycles, and there is no measurable slowdown in macrobenchmark test. We have also tested at system level, that is, interposing every system call invocation in the system and have not experienced any noticeable slowdown.

In summary, we make the following contributions:

- We first propose a whitelist based model for system call sandboxing represented as an application-specific system call table tailored to per program's needs. The approach is significantly more practical than the existing system call based sandboxing and intrusion detection systems. Many theoretic and technical challenges that impede the deployment of the sophisticated models are avoided or mitigated in our stateless model.
- We have validated our hypothesis that injected malicious code such as shellcode very often invokes some system calls outside the set used by benign programs with 134 applications against 120 shellcode.
- We have implemented a prototype including a static analyzer and the enforcement. We have demonstrated that our system can be enforced with near zero runtime overhead and it does not incur false positives in practice. It costs virtually nothing from either developers or system. Therefore, it has the potential to be deployed at system level to enhance security.

The remainder of the paper is organized as follows. We present our key observations and validate the hypothesis in §2. We then review system call models and compare the existing ones with ours in §3. We present our system design in §4 and implementation in §5 (Static Analyzer) and §6 (Enforcement). We briefly discuss deployment options in §7. The evaluation of our research is presented in §8. We discuss related work in §9 and future work in §10, and conclude in §11 .

2. KEY OBSERVATIONS

Our hypothesis is that many exploits via code injection use some system calls that are not used by the program being exploited. If this hypothesis holds, blocking the system calls not included in the whitelist of system calls of the host program is an effective way to defend exploits. Figure 1 illustrates such a situation that the exploit can be prevented by disabling any of the system calls in the shadow area without incurring denial of service in the benign program. We validate this hypothesis based on the following two observations. In §8 we present more evidences on the validity of our hypothesis.

2.1 Observation 1: Most programs only use a small set of system calls.

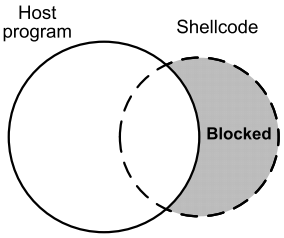

Figure 1: System calls used by a shellcode and a host program being attacked.

Table 1: System call count.

Program	Size	Count	Percentage (%)
Apache-2.2.14	1.2M	79	23.1
Bind-9.9	486K	93	27.6
Sendmail-8.14	806K	97	28.8
Finger-0.17	13K	28	8.3
Openssh-5.3	416K	38	11.3
Vsftpd-2.2.2	116K	84	24.9
Monkey-0.9.3.1	46K	49	14.5
Hypermail-2.2.0	277K	33	9.8
Grep-2.5.4	99K	17	5.0
Gzip-1.3.12	57K	25	7.4
Tar-1.22	247K	66	19.6
Openssl-0.9.8	441K	38	11.3
Bison-2.4.1	251K	21	6.2
Binutil-2.22*	70–359K	13–25	3.9–7.4
Coreutil-8.13*	0.7–4.3M	14–24	4.2–7.1
Ccrypt-1.7	44K	26	7.7
Gnupg-2.1	26K	33	9.8

*Binutil-2.22 and Coreutil-8.13 contain 16 and 103 programs, respectively. Their system call counts ranges are 13–25 and 14–24, respectively.

A commodity operating system kernel usually provides the same set of system calls to every application. For example, Linux kernel 2.6.32.59 has 337 system calls in total.[2] However, in practice no program uses the whole set of system calls. Table 1 shows the number of system call types used by 134 benchmark programs including both popular server applications and common utility programs and the percentage of used system calls over the total 337 system calls. (We present the method we used to extract system calls from executable binaries automatically in §5.) The first column lists the software name and version number; the second column is the executable size in bytes; the third column is the number of different system calls invoked by a program; the fourth column shows the percentage of the system calls used by a program. From the table, we can see that all the programs use less than 100 system calls, which is about 30% of the 337 system calls in Linux.

Figure 2 shows the percentage of the benchmark programs that use more than certain numbers of different system calls.

[2]We refer to NR_syscalls, which is defined as 337 in this version. However, there are some (21) system calls that are not implemented or deprecated, and some system calls are actually redundant in functionality. We will take this into account when measuring the effectiveness.

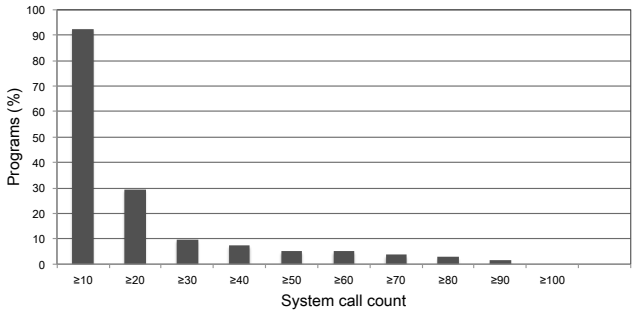

Figure 2: System call coverage.

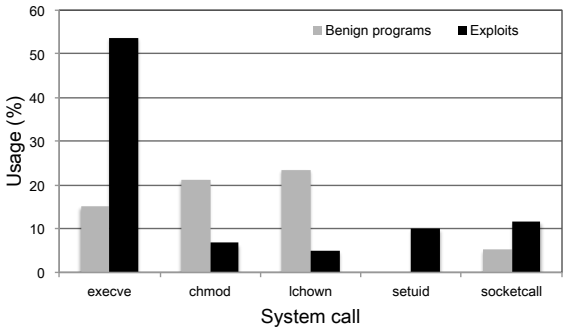

Figure 3: System call usage statistics for some dangerous system calls for benign programs and shellcode.

From the figure, we can see that over 70% of the programs use less than 20 system calls, while only 8% of the programs use more than 40 system calls.

This fact implies that, by disabling system calls not used by a host program, we may block a rich set of attack vectors if they rely on any of the disabled system calls, which is validated by our second observation.

2.2 Observation 2: The system calls not in the whitelist of a host program are viable parts of many attack vectors.

To show how shellcode in reality uses system calls, we collected 120 off-the-shelf exploits and shellcode developed by professionals from Shell-storm.org [32], a hacking organization. Table 2 shows some of the *dangerous* system calls used by shellcode, the number of their occurrences and the average shellcode sizes. It shows that the dangerous system calls such as execve and setuid are frequently used in shellcode.

We also observed that many exploits use system calls outside the white list of a host program. We have compared how each system call is used by shellcode and benign programs. The result for system calls listed in Table 2 is shown in Figure 3. Consider execve in the graph as an example, over 50% of shellcode instances invoke this system calls, while only less than 15% of the benign programs use it. This means that, by merely disabling just this system call, 85% of the benign programs can block over half of the shellcode exploits.

Table 2: The real-world shellcode.

System call	Function	Shellcode count*	Average LOC*
execve	execute any program	67	64
setuid	privilege elevation	14	18
chmod	steal password from /etc/shadow	10	23
lchown	privilege elevation	8	23
socketcall	set up backdoor connection	18	72

*Shellcode count indicates the number of shellcode instances that use that system call. The average LOC tells the average lines of shellcode in the x86 assembly language. It shows even small shellcode contains those *dangerous* system calls.

Combining Observation 1 and 2, we can see that indeed many exploits use the system calls outside the whitelist of the host program. We conclude that the whitelist model is a very effective and practical defense solution to enhance the system security. Note that our goal is not to prevent all the attacks, but to terminate a rich set of attack vectors and thus block the exploits. Actually the overall detection and prevention effect is significant according to our experiments (§8.1): 35% of the shellcode attacks can be detected by 90% of the benign programs, while 80% of the benign programs can prevent 70% of the exploits.

3. SYSTEM CALL INVOCATION MODELS

3.1 Existing Models

A lot of models have been proposed [12, 31, 35, 14, 10] for the intrusion detection systems based on modeling system call behaviors. These models can be roughly categorized into two types: (1) Finite State Automata (FSA) and (2) Pushdown Automata (PDA). By training or static analysis, an automaton is established to model the normal program behaviors in terms of system call requests. Each system call triggers a state transition, which is verified against the model; if the transition is not accepted by the automaton, an attack is reported.

There are a diversity of FSA models, which are constructed either by training or by static analysis. The training-based models learn valid transitions through exercise. The common limitation is that false alarms may occur if some legal execution paths are not gone through during training [12, 31, 10]. On the other hand it is difficult to construct a precise FSA model via static analysis. A lot of challenges and questions, for example, how to deal with signal handling and multithreaded programs, and setjmp/longjmp, have not been well resolved [31, 14, 15, 13]. In addition, most implementations trace the system call requests using the inter-process system call tracing mechanism, which typically incurs high running time overhead. Although some work states that a kernel-space implementation can improve the efficiency [31, 35], the expected overhead is still high. For example, the models usually rely on mapping the system call back to the call site of the original program to determine the state transition, which requires traversing over the user space call stack and thus incurs significant overhead. To the best of our knowledge, none of the models has been widely applied and deployed in practice due to the various limitations and issues.

In addition to modeling system call sequences, the PDA further captures the calling context information and makes sure the return addresses on the stack are not corrupted, so that it is more difficult for attackers to evade the detection. This model is prohibitively expensive, typically with minutes, sometimes hours, of overhead per transaction, e.g., sending an email [35]. Although Giffin et al. reported high efficiency in their experiment [14], as pointed out in [10], it is due to the context of remote execution and the low overhead cannot be expected under the host-based intrusion detection environment.

3.2 Mimicry Attack and the whitelist Model

Instead of putting forward a more advanced model by tracing the program behavior more precisely and thus introducing even higher overhead, we propose a "degraded" model—a whitelist of system calls, which can be regarded as an extremely simple FSA which accepts the set of system call sequences Σ^*, where Σ is the set of valid system calls used by the application in consideration. The insights include (1) a static analysis on binaries for extracting the whitelist is more feasible compared to constructing a FSA and PDA(§5); (2) the whitelist model can be easily implemented as an application-specific system call table (§6.2), which is then enforced with near zero overhead; (3) more importantly and surprisingly, the "degraded" model does not really degrade the detection effect for typical applications compared to the sophisticated FSA/PDA model, which is elaborated as follows.

A typical attack can be divided into a penetration phase when the attacker hijacks the control flow and an exploitation phase when the attacker exploits the control of the application to harm the rest of the system [36]. Most anomaly detection systems are based on detecting the harmful effects shown in the exploitation phase, rather than detecting the penetration itself. For example, a buffer overflow attack overwrites a function pointer or a return address to alter the execution path without issuing any system calls; thus, no syscall-based IDS can detect the buffer overflow itself. After hijacking the control flow, the attacker can force any system call sequence accepted by an FSA model. Given an attack consisting of a specific system call sequence, the attacker can simply pass effectively *null* arguments to system calls of no interest, which is referred to as *no-op* system calls. for example, open(null, 0) if open is not needed, and pass the arguments she needs when a system call in the malicious sequence is reached. The attacker can synthetically construct the malicious sequence by mixing the no-op and attacking system calls, which is called *mimicry attacks* [36]. An ap-

Table 3: Model comparison.

Model	whitelist	FSA	PDA
Model Construction	Easy	Difficult	Difficult
False alarms	Few	Various	Various
Enforcement overhead	Near zero	High	Very high
Security	Good	Good	Good

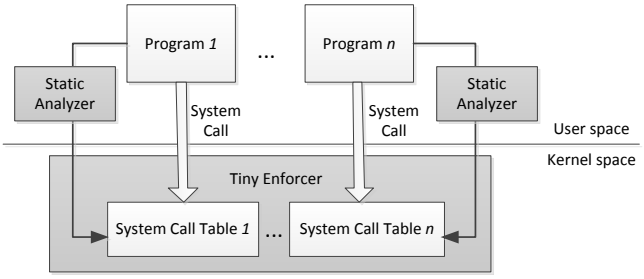

Figure 4: Architecture.

plication usually has an outer loop implementing the major functionality. For example, a web server repeatedly services incoming requests and a database server keeps processing the queries. The majority system calls in the whitelist of a given application should be contained in the loop. Therefore, one can expect that most system call that can be found in the whitelist may become reachable with a mimicry attack, which renders the FSA model effectively a whitelist in the perspective of attackers. The implications are two folds: some attacks that were deemed detectable can evade an FSA via mimicry attacks; on the other hand, a wide range of attacks that can be detected by an FSA even after applying mimicry attacks can be detected using the much simpler whitelist model.

3.3 Model Comparison

We summarize the comparison of FSA, PDA, and the whitelist models in Table 3. The model construction for the whitelist is much easier with few false positives; we have not encountered any false positives in our experiments. We have near zero model enforcement overhead, while for FSA and PDA the runtime overhead is high to very high. In summary the whitelist model is feasible to be constructed and achieves good detection effectiveness with virtually zero overhead.

4. SYSTEM OVERVIEW

Our system, named System Interface Tailor (SIT), as shown in Figure 4, consists of two components, the Static analyzer and the Tiny Enforcer. The Static Analyzer, given a program binary, discovers its whitelist. When a program is executed, the Tiny Enforcer creates an application-specific system call table based on the program's whitelist. For each entry in the system call table, if the corresponding system call belongs to the whitelist, the entry is filled out with the original handler function's address; otherwise, it points to an alarm function. During runtime, the Tiny Enforcer interposes each system call request and dispatches the request using the current process's system call table.

The Static Analyzer disassembles the binary to do an interprocedure analysis to extract the system calls needed by the target binary. Although the whitelist model avoids some common difficulties faced by the stateful models, the analysis has to deal with challenges such as indirect procedure calls, implicit and explicit dynamic linking, and special system call invocations (§5).

After comparing the design choices for enforcing the model, we present the Tiny Enforcer, which encodes the whitelist model as a per-process system call table in kernel space(§6). The design incurs minimum time and memory overhead.

The whitelist generated via binary static analysis has to be stored at a safe place. The deployment and software updating issues will be discussed in §7.

We assume the Linux operating system on the x86 platforms when discussing the implementation details, although all the techniques presented are applicable to many other well-known hardware and operating systems, e.g., Solaris and FreeBSD. We target applications of the ELF binary format which are popular on Unix platforms.[3]

5. STATIC ANALYZER

Since it is much easier to obtain the system calls used in a statically linked program, we focus on dynamically linked programs, which are more common in modern operating systems. We first briefly introduce the methods of invoking a system call provided by Linux on the x86 platforms. Then we present, given a program, how to discover the system calls issued via dynamic library functions, and apply the techniques of data flow analysis to deal with some special cases. We describe some implementation issues at the end of this section.

5.1 Background

The Linux kernel provides two different ways for user programs to issue a system call.

int 80h: it issues a system call through the 0x80 interrupt.

sysenter: the `sysenter` instruction was introduced in the Intel Pentium II CPU to speed up the switch from the user mode to kernel mode and has been supported since the Linux 2.5 kernel.

A user program saves the ID of a system call to be invoked in the `eax` register and executes one of the two instructions above to issue a system call. The net effect of either of the two instructions is a jump to a function called *system call handler*. The kernel maintains a system call table, which is an array of function addresses with the ith entry containing the *service routine* address of the system call having the ID number i. The system call handler uses the system call ID saved in `eax` as an index to locate the address of the service routine in the system call table and invoke it.

5.2 Interprocedural Analysis for System Call Discovery

[3]Our system will be open sourced and available at `http://anonymized-url/`.

In this section we present the interprocedural analysis for system call discovery. Instead of applying the low level primitives above, programs usually rely on the *libc* library to invoke system calls. The libc library contains *wrapper routines* whose only purpose is to issue system calls. It is trivial to establish a one-to-one mapping from a wrapper to a system call. We then perform an interprocedural analysis from the program binary to search all the wrappers reachable.

An ELF format binary file contains one dynamic linking section that list all the dynamic library functions referenced by the binary directly. These functions become the entry points of the interprocedural analysis. The algorithm consists of the following steps.

1. Three data structures are maintained: two sets, the Processed Function Set (the functions that have been processed) and the whitelist Set (the system call whitelist), and one queue, which contains functions to be processed. Initially, the whitelist Set is empty, while the Processed Function Set and the queue contain the functions listed in the program binary file's dynamic linking section.
2. If the queue is empty, the algorithm ends. Otherwise, one node is dequeued from the queue as the current function. The algorithm goes to the next step.
3. If the current function is a libc wrapper routine, it is added into the whitelist Set. Otherwise, the interprocedural analysis is performed starting from the function. *The analyzer simulates the dynamic linker to locate the library defining the current function and generates a control flow graph for it.* For each of the functions called by the current function, if it is not an element of the Processed Function Set, the callee function associated with the library defining the function is enqueued into the queue and added into the Processed Function Set. Go back to Step 2.

When the algorithm ends, the system calls needed by the given program are obtained from the whitelist Set.

5.3 Indirect Procedure Calls

When dealing with indirect procedure calls via, e.g., function pointers or dynamic dispatching, we consider the following cases and solutions.

First, if the indirect procedure calls reside in the program's binary file, the dynamic linking library functions referenced by the invocation are contained in the dynamic linking section of the binary file, so they are processed as above.

Second, a function pointer is usually used for callback purposes, which is also the main usage of function pointers. For example, by passing a custom function to the sort function, e.g., qsort, one can affect how to sort the data. the program can also pass functions to signal or sigaction to register custom signal handlers. The functions passed are usually contained in the binary file. So the library functions referenced by these functions are also listed in the dynamic linking section.

Third, we rely on points-to analysis, which is a well-researched problem, to do a conservative interprocedural analysis using 0CFA [33]. While the targets of indirect calls are not decidable in theory, substantial research in this area has shown reasonable results [17, 38, 24].

Based on the observations and solutions above, we are able to discover the majority of the whitelist. Finally, we propose to use a runtime remedy component to deal with incomplete whitelist. When an invocation to a system call outside the whitelist is detected, we pause the program and traverse the integrity of the call stack and the jump table of all the libraries, and make sure every return address and function address points to binary or library code pages. Once one address pointing to heap or stack data is found, an attack is detected. Otherwise, we add the current system call into the whitelist. Although this kind of checking pauses the program execution, it occurs very occasionally, and thus will not affect the performance much in practice. We have not encountered incomplete whitelist issues in the experiments. The remedy solution may incur false positives in the case of self-modifying code, which is not addressed in this research.

5.4 Special Cases

While system calls are generally issued via regular library function calls, the following special cases have to be taken into account:

- Invoking system calls through sysenter or int 80h directly. Some C programs contain inline assembly code which is capable of using these instructions directly. The value saved in the eax register when issuing the low-level primitive is the system call ID.
- Explicit dynamic linking via dlopen/dlsym. Some applications determine the library to be referenced from a set of alternatives during runtime. Our principle is to consider all the alternative libraries regardless of which will be used finally. The first argument passed to dlsym is the handle of the opened library which is returned by a call to dlopen, while the second one indicates the name of the function to be used.
- Issuing system calls through libc function syscall, which accepts the system call ID as the first argument.

While it is straightforward to identify the patterns above, how to recover the arguments (including the value saved in the eax register) needs special handling. Once the arguments in a pair of dlopen and dlsym are recovered, we can apply the algorithm described in §5.2 to discover the referenced system calls.

Therefore the critical task is to, given an argument variable used in those special cases, determine the possible values of the argument. We use interprocedural data flow analysis techniques to solve this problem in three stages[29, 30]. (1) We compute the use-def chain [2] in terms of the argument variable and intend to calculate its possible values. If the values can be determined by following the use-def chain, the algorithm stops. Otherwise, if the values depends on the formal arguments of the containing function, we generate a *data*

flow summary function that describes the relations between the function's formal arguments and the target variable. (2) We search all the calls to that function and continue data flow analysis at the caller functions as the first stage. This stage can be recursive. (3) By composing the data flow function summary with the knowledge of use-def, we can recover the argument values that are statically known.

If the argument values depends on external input or it is too complicated to generate precise function summaries, the argument recovery will fail. The runtime remedy component can be used to deal with the incompleteness of the whitelist of undecidable arguments.However, generally the arguments passed as the system call ID, the library or function names are constant, static values.

5.5 Avoided Challenges

Multithreading: The threaded program is another challenge for FSA model enforcement. Specifically, the model enforcer usually resides in another process, so that the thread context-switch events, which normally imply exceptional control flow change, are not received by the enforcer and thus may be viewed as deviations from the model. Our enforcer resides in the kernel and can interpose such events conveniently.

Signal handling: The signal handling also introduces exceptional control flow change, which is hard to be modeled using FSA, while our whitelist model is not control flow sensitive and the signal handler functions are processed as other functions.

6. ENFORCEMENT

We first compare different design choices and explain why we choose kernel-space implementation. Then we present the virtual application-specific system call table.

6.1 Why Kernel-space?

As there are a variety of approaches to enforcing the whitelist model, we will compare various designs and demonstrate that the kernel-space enforcement is promising in terms of robustness and efficiency. We believe the two requirements are most important. In an adversary setting or given a flawed and buggy program, it is desirable to guarantee the enforcement itself is not corrupted. The runtime overhead due to the security enhancement should be low.

System call monitoring in user space has been well explored [21, 19]. It usually replaces the default libc library with another library to add additional functionality, e.g., buffer overflow checking. To enforce our model, we can rewrite the wrappers forwarding or denying the libc calls according to the whitelist. The implementation is straightforward and the overhead is low. However, it assumes all the system calls are issued via libc function calls, which is not the case. In particular, once the control flow is hijacked, the attacker can issue any system call via traps evading the wrappers.

Intrusion detection systems based on system call monitoring [12, 35, 15, 31, 10, 15] usually trace the execution of the target process in a separate process. When a system call is issued in the target process, the kernel pauses its execution and notifies the monitor process, which responds according to the policies or the model and switches back to the kernel and resumes the execution of the target process. This incurs an overhead of multiple context switches per system call, which renders IDSes based on this system call tracing mechanism very inefficient and thus impractical.

By enabling Linux Security Modules (LSM), we can implement the model as a security module. However it introduce system-level overhead, although it is not significant. More importantly LSM exports all of its symbols; it facilitates the insertion of malicious modules, e.g., rootkits, as well as security modules.

Hypervisor-based system call hooking does not require changes to the kernel, which is particularly useful when the kernel code is not available. However, the interposition also involves extra context switches between the hypervisor and the kernel. The scalability is also problematic due to the need of maintaining the mapping between the processes and their whitelists in the typically small hypervisor memory space. After all, it violates the principle of the hypervisor to enforce high-level semantics. Nevertheless, we regard hypervisor-based approach as a viable alternative when the kernel code cannot be accessed or modified.

We note that each commodity operating system provides only a few mechanisms for issuing system calls from user space. Each entry of the mechanism takes the system call number to index the system call table, dispatching the handling. We finally decide to use an application-specific system call table based on the whitelist for dispatching. The details are elaborated in §6.2. The overhead is virtually zero according to our evaluation. Each system call request has to pass the enforcement before it is serviced. Therefore, the enforcement cannot be evaded from user space. On the other hand, if the kernel has been compromised, the enforcement can also be undermined. However, in that case, the attacker does not need to rely on system calls to exploit at all. While how to protect the kernel is a separate topic, it is noticeable that by confining the set of system call that can be issued from a program, it usually helps enhance the system security as it imposes extra obstacles against attackers. For example, an attacker is not able to escalate the privileges (setuid) and install kernel modules containing rootkits (create_module) if either of the unusual system calls does not belong to the whitelist of the compromised process.

6.2 Virtual App-specific System Call Table

A straightforward implementation for the kernel-space enforcement is to generate a separate system call table for each process (or application) and fill out each entry with the real system call handler's function address if this system call belongs to the process's whitelist, or a common alarming function's address, otherwise. However it may increase the pressure of data cache and imposes kernel memory overhead.

We, instead, propose a more compact representation, named *Virtual Application-specific System Call Table*, which is a bitmap saved directly in the process descriptor. Each entry (bit) in the bitmap corresponds to an entry in the system

call table. If a system call belongs to the whitelist, the corresponding bit in the bitmap is 1; otherwise, 0. There is only one copy of the real system call. The enforcement logic is to, given a system call, check the bitmap of the current process. If the bit indexed by the system call number is 1, the system call is dispatched using the real system call table as normal. Otherwise, the alarming function is invoked.

Each Virtual Application-specific System Call Table only needs to occupy around $N/32$ words on 32-bit OSes, where N is the total number of system calls. That is, less than 12 words on current Linux for most platforms. Thus it incurs minimum memory overhead and virtually zero extra pressure on data cache. The Virtual Application-specific System Call Table design involves one more memory read to retrieve the containing word from the bitmap. However, since process descriptor probably has to be read into the data cache during system call handling, the one more memory read can be counted as one data cache read.

In addition, as in Linux each thread has its own process descriptor (task_struct), this design simply fits the multithreaded programs by putting a copy of 12-word bitmap into each thread's task_struct, compared to maintaining a reference counter to the process's single system call table. The per-thread Virtual System Call Table is also more extensible to enforce a per-thread system call whitelist.

6.3 Dealing with fork/exec

When a new thread or process is forked due to a fork or clone system call, the new task_struct instance will be copied from the parent's inside do_fork, so that the new task_struct inherits the virtual system call table from the parent.

When an execve system call is made to run a new program, we retrieve its whitelist and fill out the bitmap according to it. If the whitelist does not exist, the program must be *foreign* code and an intrusion detection is reported. How to save and retrieve the whitelists is a deployment issue and is discussed in §7.

7. DEPLOYMENT

The deployment of SIT involves a safe store of the whitelist and a patch and update of the kernel. Our static analyzer can go through all the programs and generate a whitelist for each of them with a one-time effort. The whitelists are then saved in a protected *registry* file like the passwd file for later reference. The whitelist needs to be re-generated whenever a program or a dependent shared object is updated. This deployment can also defeat some trojan attacks that replace existing programs on the system, since the whitelist may not fit the malware.

8. EVALUATION

8.1 Effectiveness

We implemented our static analysis component as the plugins on IDA Pro [18], a professional disassembler and reverse engineering tool. Figure 5 shows the detection ratio of our benchmark programs against the shellcode. There are two sets of shellcode. One is the original shellcode, while the other is the modified shellcode, taking into account of equivalent system calls, which is listed in Table 4.[4] We assume attackers may revise the shellcode to circumvent our system by trying to use equivalent system calls included in the whitelist of the victim program. We thus calculate the prevention ratio in the case of modified shellcode as follows. If, for example, a benign program uses chdir, we add both chdir and fchdir into the whitelist.

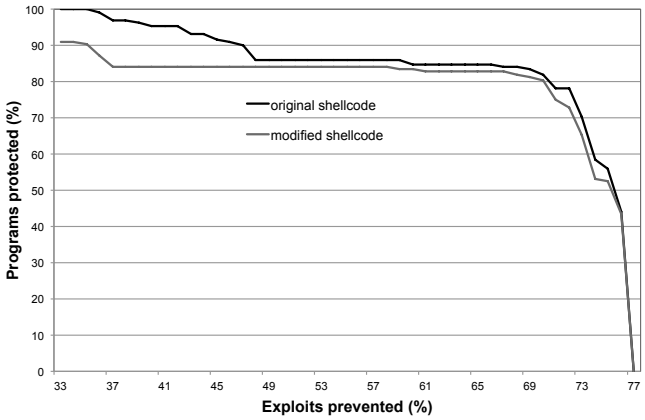

Figure 5: Shellcode detection ratio.

From the Figure 5, we can see that our system can prevent about 35% of the original shellcode on all the benign programs, and 70% of the original shellcode are prevented by about 82% of the benign programs. In the case of modified shellcode the prevention ratio degrades a little. 90% and 80% of the benign programs can prevent 35% and 70% of the modified shellcode, respectively. This shows that our system is very effective in protecting benign programs from shellcode exploits.

8.2 Correctness

To evaluate whether our system incurs false positives or denial of services in practice, we ran the benchmark programs in Table 1, in total 134 benign programs. For utility programs such as ls and objdump, we use the test input of the distribution and other random input data. For server programs like bind/named, we deploy them online and send requests for a period of time. The testing time ranges from a few seconds to days. During our evaluation period, we did not see any false positive.

8.3 Efficiency

We evaluated the runtime efficiency on a Dell Precision Workstation T5500 with two 2.26GHZ Intel Xeon E5507 quad-core processors and 4GB of RAM running 32-bit Ubuntu 10.04 with Linux kernel 2.6.32.59. The memory overhead is constantly 12 words for each thread due to the bitmap for the 337 system calls and the data alignment. We focused on time overhead.

8.3.1 Microbenchmark

Table 4: Equivalent system call sets

Function	System calls
change working directory	chdir, fchdir
reposition read/write file offset	lseek, llseek
map files or devices into memory	mmap, mmap2
create pipe	pipe, pipe2
create a child process	fork, vfork
open a file	open, openat
delete a name the file refers to	unlink, unlinkat
unmount filesystems	oldumount, umount
duplicate a file	dup, dup2, dup3
read value of a symbolic link	readlink, readlinkat
synchronize a file's state on disk	fsync, fdatasync
get list of supplementary group IDs	getgroups16, getgroups
set list of supplementary group IDs	setgroups16, setgroups
get real group ID	getgid16, getgid
get real user ID	getuid16, getuid
get effective group ID	getegid16, getegid
get effective user ID	geteuid, geteuid16
get real, effective and saved user ID	getresuid16, getresuid
get real, effective and saved group ID	getresgid16, getresgid
set real user ID	setuid16, setuid
set real group ID	setgid16, setgid
set real and effecive user ID	setreuid16, setreuid
set real and effecive group ID	setregid16, setregid
set real, effecive and saved user ID	setresuid16, setresuid
set real, effecive and saved group ID	setresgid16, setresgid
send signal to a process or process group	kill, tkill, tgkill
enter virtual 8086 mode	ptregs_vm86old, ptregs_vm86
retrieve an extended attribute value	getxattr, lgetxattr, fgetxattr
list extended attribute names	stxattr, llistxattr, flistxattr
get information about current kernel	olduname, uname, newuname
read from a file	read, readv, pread64, preadv
wait for process to change status	wait4, waitpid, waitid
synchronous I/O multiplexing	old_select, select, pselect6
change permissions of a file	chmod, fchmod, fchmodat
set an extended attribute value	setxattr, lsetxattr, fsetxattr
change times of an inode	utime, utimes, futimesat, utimesat
write to a file	write, writev, pwrite64, pwritev
make a new name for a file	link, linkat, symlink, symlinkat
create a special or ordinary file	mknod, mknodat, mkdir, mkdirat
get file system statistics	statfs, fstatfs, statfs64, fstatfs64
remove an extended attribute	removexattr, lremovexattr, fremovexattr
truncate a file to a specified length	truncate, ftruncate, truncate64, ftruncate64
change ownership of a file	chown, fchown, lchown, chown16, fchown16, lchown16, fchownat
get file status	stat, lstat, fstat, newstat, newlstat, newfstat, stat64, lstat64, fstat64, fstatat64

Table 5: Microbenchmark tests.

	getpid	system
Without SIT	0.081μs	1,109μs
With SIT	0.084μs	1,124μs
Overhead	~6.5 cycles	15μs

We ran the system call getpid (issued using sysenter) and the libc function system for one million times each with and without the enforcement, respectively. The average times for each call are presented in Table 5. The overhead for each getpid call, which counts for 6.5 CPU cycles on our test machine, is due to a bitmap read and a cmp-jmp instruction combination inlined in per system call handling. The overhead (1.4%) for each system invocation represents the slowdown due to the whitelist retrieval and system call table assembling when starting a new process.

8.3.2 Macrobenchmark

We ran SPEC CPU2006 Integer benchmark suite, which comprises a variety of compute intensive programs including perlbench, bzip2, h264ref, etc. The scores (13.4) with and without our enforcement are the same. We also measured the time for compiling the Linux kernel 2.6.32.59 with the concurrency level set as the number of CPU cores using the command time make -j8 bzImage, and the overhead is also zero.

9. RELATED WORK

We break down the related work into two groups: (a) sandboxing, and (b) intrusion detection systems based on system call models.

Sandboxing. Since the seminal work of Software Fault Isolation (SFI) [37], substantial research has been done on confining the problematic operations of faulty, untrusted, or compromised program components. These efforts could be categorized as follows. (C1) Software-based fault isolation mechanisms, such as [8, 5, 26], focus on confining kernel modules, e.g., kernel drivers, through code annotations and rewriting to protect other parts of the kernel. Our work intend to prevent compromised applications from harming the rest of the system. (C2) System call interposition for confinement. MAPbox [1] groups application behaviors into classes based on their expected functionality and grants the resources required to achieve that functionality. REMUS [3] defines the rules with respect to valid argument values passed to system calls and enforces the rules by instrumenting the implementation of the system call service routines in a loadable kernel module. SubDomain [7] allows the administrator to specify the *least* privileges needed by a program and enforces it using the same technique as REMUS. Vx32 [11] allows an application to host untrusted guest plug-ins in a safe way by defining confined virtual system calls that can be invoked by the plug-ins. Goldberg et al. proposed to, given an application, manually identify *dangerous* system calls, which are filtered out by tracing the system call invocations of the target program in a separate process using ptrace [16]. Systrace [28] generates policies based on both training and configuration. Its enforcement combines both the kernel instrumentation and an inter-process system call tracing. (C3) Hardware-based approaches rely on specific hardware support, e.g., tagged memory [40], or special hardware not available in any existent processor [39] to enforce security policies. (C4) Language-based approach. Singularity [9] is an experimental operating system that achieves strong isolation and controlled communication leveraging type-safe languages such like Sing#, an extended version of Spec#, which itself is an extension of C#). (C5) Virtual machine based isolation. Apiary [27] isolates desktop applications in ephemeral containers. Overshadow [6] and Proxos [20] enforce security policies by interposing kernel APIs from a hypervisor. Mechanisms in this category usually incur significant overhead.

Our proposal belongs to class C2. Two main differences distinguish our proposal from existing C2 techniques. (1) Most C2 techniques require manual configuration to define the policies and rules, for example, specifying the set of *dangerous* system calls for the target application, which is error-prone, time-consuming, and may lead to false positives, while our static analyzer identifies the set of invalid system calls rigorously and automatically. (2) Compared to techniques that enforce their policies at the cost modest to significant runtime overhead, our approach is much *simpler* in both logic and implementation and leads to virtually no overhead (6.5 CPU cycles per system call checking on our test machine) based on the bitmap-based virtual system call table.

Intrusion detection. Host-based intrusion detection systems that detect anomaly based on system call models have evolved from the earliest N-gram system call sequence models [12] to FSA [31, 35] and PDA [35] models. They share similar obstacles with the system call level sandboxing techniques regarding precise model construction and effective enforcement mechanisms [12, 31, 35]. Our whitelist model is much simpler, with virtually no developer cost on model construction, near zero runtime enforcement overhead, practically no false positives or denial of services.

As detailed in §3, the mimicry attack transforms a malicious attack sequence to a cloaked seemingly valid sequence by, e.g., inserting *no-op* system calls, evading FSA models [36]. Kruegel et al. [22] further demonstrates that even a PDA model can be evaded by counterfeiting the call stack before issuing a system call and regaining the control flow after the system call returns. Their work illustrates that launching a mimicry attack is easy [36] and can be automated [22]. Consequently, both in theory and practice the effect of a stateful FSA or PDA model is actually similar or equivalent to that of a stateless model, that is, a whitelist of system calls. This insight leads us to this work in order to show the *net* detection effect of the FSA or PDA model-based IDS and the efficiency it can achieve with our dramatically simplified model.

To better detect the mimicry attack, two branches of efforts have been made. Some work intends to capture more information on the call stack [31, 10, 15], which has been proven ineffective [22]. Others exploit the environment vari-

[4]Note that we do not really modify the shellcode.

ables, configuration files, and the command input to discover constraints on the arguments passed to the system call at each call site [35, 13, 4]. They are usually based on FSA or PDA models. While different argument rules are associated with different call sites, with the mimicry attack an adversary can again freely choose the call site. In other words, the argument verification does not benefit from the complex and heavy-weight FSA model by distinguishing different call sites. Considering the mimicry attack, if we apply the argument constraints to the whitelist model, we can expect a similar detection effect with much less overhead in runtime and complexity in terms of the model generation.

10. FUTURE WORK

Applying intrusion characterizing techniques to the whitelist model. Based on the fact that the mimicry attack that can be easily launched by attackers, which renders the state-based models stateless in detection effect, a lot of intrusion detection techniques previously based on the sophisticated and costly FSA or PDA models can be deployed on the highly efficient application-specific system call based intrusion scheme. For example, techniques that learn or discover system call argument constraints can work with the whitelist model to further confine the system call invocations [35, 4]. Another example is that the white list model can be further parametrized with the command line input, environment variables, and configuration files, so that it can be further tailored as some execution paths may be pruned due to the runtime environment [13]. These techniques have been proven effective but are not widely deployed mainly due to the complicated and inefficient underlying models. Our simple but effective model launches new possibilities for applying these techniques.

Per-thread system call table. Each thread actually owns a virtual system call table (a 12-word bitmap). Currently, all threads belonging to the same process have the same system call table. However, a lot threads only run a relatively short piece of code and therefore, we can further confine the program behaviors by enforcing system call tables tailored for each thread.

Mapping high-level system functionality to a few bits. With the per-process or per-thread system call table, we can map each high-level system service, for example, network access and device manipulation, to a few system calls. Users specify which services should be disabled for a specific program; accordingly, we can reset the corresponding bits of the services.

11. CONCLUSION

The insight that the mimicry attack actually renders the complicated and inefficient FSA and PDA models as stateless in terms of detection effect has motivated the exploration of the simple whitelist model, which can be enforced as a compact per-process virtual system call table. Due to the simplicity of the model we are able to construct a precise whitelist with practically no or few false positives. Its enforcement imposes near zero runtime overhead. It is certainly not a panacea for all the code injection attacks, however, the experiments have shown that the model is effective for defeating a significant number of exploits. Our system also provides a much more feasible approach to applying the ideas previously working with FSA and PDA, such as argument constraints, to further enhance application and system security.

12. REFERENCES

[1] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security '00*, 2000.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] M. Bernaschi, E. Gabrielli, and L. Mancini. REMUS: A security-enhanced operating system. *ACM TISSEC*, 2002.

[4] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *IEEE Symposium on Security and Privacy*, 2006.

[5] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-granularity Software Fault Isolation. In *SOSP '09*, 2009.

[6] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS '08*, 2008.

[7] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *LISA*, 2000.

[8] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI '06*, 2006.

[9] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys '06*, 2006.

[10] 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*, 2003.

[11] B. Ford and R. Cox. Vx32: Lightweight userlevel sandboxing on the x86. In *USENIX ATC*, 2008.

[12] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, 1996.

[13] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In *the 8th international conference on Recent Advances in Intrusion Detection*, 2005.

[14] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *the 11th USENIX Security Symposium*, 2002.

[15] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium*, 2004.

[16] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *USENIX Security Symposium*, 1996.

[17] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. OOPSLA '97, pages 108–124.

[18] Hex-Rays. IDA Pro. `http://www.hex-rays.com/products/ida/index.shtml`.

[19] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *IEEE Symposium on Security and Privacy*, 1997.

[20] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *IEEE Symposium on Security and Privacy*, 1997.

[21] E. Krell and B. Krishnamurthy. COLA: customized overlaying. In *Usenix Winter Conference*, 1992.

[22] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security*, 2005.

[23] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *ESORICS*, 2003.

[24] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. POPL '11, pages 3–16.

[25] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 2010.

[26] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *SOSP '11*, 2011.

[27] S. Potter and J. Nieh. Apiary: Easy-to-use desktop application fault containment on commodity operating systems. In *USENIX ATC*, 2010.

[28] N. Provos. Improving host security with system call policies. In *USENIX Security Symposium*, 2003.

[29] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL '95, pages 49–61.

[30] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected papers from the 6th international joint conference on Theory and practice of software development*, TAPSOFT '95, pages 131–170, 1996.

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

[32] Shell-Storm.org. A development organization based on GNU/Linux systems, 2012. `http://www.shell-storm.org/shellcode/shellcode-linux.php`.

[33] O. Shivers. Control flow analysis in scheme. PLDI '88, pages 164–174, 1988.

[34] G. Tandon and P. Chan. Learning Rules from System Call Arguments and Sequences for Anomaly Detection. In *ICDM Workshop on Data Mining for Computer Security (DMSEC)*, 2003.

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

[36] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *the 9th ACM conference on Computer and communications security*, 2002.

[37] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *SOSP '93*, 1993.

[38] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. PLDI '04, pages 131–144, 2004.

[39] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. In *SOSP '05*, 2005.

[40] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI 2008*, pages 225–240. USENIX Association, 2008.