

Automatic Extraction of Accurate Application-Specific Sandboxing Policy

Lap-chung Lam and Tzi-cker Chiueh

Rether Networks, Inc.
99 Mark Tree RD Suite 301, Centereach NY 11720, USA
lclam@cs.sunysb.edu
<http://www.rether.com>

Abstract. *One of the most dangerous cybersecurity threats is control hijacking attacks, which hijack the control of a victim application, and execute arbitrary system calls assuming the identity of the victim program's effective user. System call monitoring has been touted as an effective defense against control hijacking attacks because it could prevent remote attackers from inflicting damage upon a victim system even if they can successfully compromise certain applications running on the system. However, the Achilles' heel of the system call monitoring approach is the construction of accurate system call behavior model that minimizes false positives and negatives. This paper describes the design, implementation, and evaluation of a Program semantics-Aware Intrusion Detection system called Paid, which automatically derives an application-specific system call behavior model from the application's source code, and checks the application's run-time system call pattern against this model to thwart any control hijacking attacks. The per-application behavior model is in the form of the sites and ordering of system calls made in the application, as well as its partial control flow. Experiments on a fully working Paid prototype show that Paid can indeed stop attacks that exploit non-standard security holes, such as format string attacks that modify function pointers, and that the run-time latency and throughput penalty of Paid are under 11.66% and 10.44%, respectively, for a set of production-mode network server applications including Apache, Sendmail, Ftp daemon, etc.*

1 Introduction

Many computer security vulnerabilities arise from software bugs. One particular class of bugs allows remote attackers to hijack the control of victim programs and inflict damage upon victim machines. These *control hijacking* exploits are considered among the most dangerous cybersecurity threats because remote attackers can unilaterally mount an attack without requiring any special set-up or any actions on the part of victim users (unlike email attachment or web page download). Moreover, many production-mode network applications appear to be rife with software defects that expose such vulnerabilities. For example, in the most recent quarterly CERT Advisory summary (03/2003) [4], seven out of ten

vulnerabilities can lead to control hijacking attacks. As another example, the notorious SQL Slammer worm also relies on control hijacking attacks to duplicate and propagate itself epidemically across the net.

An effective way to defeat control-hijacking attacks is application-based anomaly intrusion detection. An application-based anomaly intrusion detection system closely monitors the activities of a process. If any activity deviates from the pre-defined acceptable behavior model, the system terminates the process or flags the activity as intrusion. The most common way to model the acceptable behavior of an application is to use system calls made by the application. The underlying assumption of the system call-based intrusion detection is that remote attackers can damage a victim system only by making malicious system calls once they hijack a victim application. This assumption is not always true, because it is possible to devise attacks that only corrupt the victim application's in-memory data structures, and eventually cause the application to misbehave in a way that benefits the attackers. Given that system call is the only means to inflict damage, it follows logically that by closely monitoring the system calls made by a network application at run time, it is possible to detect and prevent malicious system calls that attackers issue, and thus protect a computer system from attackers even if some of its network applications have been compromised. While the mechanics of system call-based anomaly intrusion detection is well understood, successful application of this technology requires an accurate system call model that minimizes false positives and negatives.

Wagner and Dean [22] first introduced the idea of using compiler to derive a call graph that can capture the system call ordering of an application. At run time, any system call that does not follow the statically derived order is considered as an act of intrusion and thus should be prohibited. A call graph derived from a program's control flow graph (CFG) is a non-deterministic finite-state automaton (NFA) due to such control constructs as if-then-else and function call/return. The degree of non-determinism determines the extent to which mimicry attack [23] is possible, through so-called impossible paths [22]. This paper describes the design, implementation, and evaluation of a Program semantics-Aware Intrusion Detection system called *Paid*, which consists of a compiler that can derive a deterministic finite-state automaton (DFA) model which captures the system call *sites*, system call *ordering*, and *partial control flow* from an application's source code, and an in-kernel run-time verifier that compares an application's run-time system call pattern against its statically derived system call model, even in the presence of function pointers, signals, and setjmp/longjmp calls. *Paid* features several unique techniques:

- *Paid* inlines each system call site in the program with its associated system call stub so that each system call is uniquely labeled by the address of its corresponding `int 0x80` instruction,
- *Paid* inlines each call in the application call graph to a function having multiple call sites with the function's call graph, thus eliminating the non-determinism associated with the exit point of such functions,
- *Paid* introduces a `notify` system call that its compiler component can use to inform its run-time verifier component of information that cannot be determined statically such as function pointers, signal delivery, and to eliminate

- whatever non-determinism that cannot be resolved through system call inlining and graph inlining, and
- *Paid* inserts random null system calls (which are also part of the system call graph) at compile time and performs run-time stack integrity check to prevent attackers from mounting mimicry attacks.

The combination of these techniques enables *Paid* to derive an accurate DFA system call graph model from the source code of application programs, which in turn minimizes the run-time checking overhead. However, the current *Paid* prototype has one drawback: it does not perform system call argument analysis. But we will include this feature in the next version of *Paid*.

2 Related Work

2.1 System Call-Based Sandboxing

Many recent anomaly detection systems [22, 8, 18, 13, 9, 24, 15, 17] defines normal behavior model using run-time application activities. Although such systems cannot stop all attacks, they can effectively detect and stop many control hijacking attacks. Among these systems, system call pattern has become the most popular choice for modeling application behavior. However, simply keeping track of system calls may not be sufficient because it cannot capture other program information such as user-level application states.

Wagner and Dean's work [22] advocated a compiler approach to derive three different system call models, callgraph model (NFA), abstract stack or push-down automaton model (PDA), and digraph model. Among all three models, the PDA model, which models the stack information to eliminate the impossible paths, is the most precise model, but it is also the most expensive model. *Paid*'s DFA model represents a significant advance over their work. First, *Paid* uses `notify` system call, system call inlining, and graph inlining to reduce the degree of non-determinism in the input programs. Second, *Paid* uses stack integrity check and random insertion of null system calls to greatly raise the barrier for mimicry attacks. Third, *Paid* is more efficient than Wagner and Dean's system in run-time checking overhead. For example, for a single transaction, their PDA model took 42 minutes for `qpopper` and more than 1 hour for `sendmail`, whereas *Paid* only takes 0.040679 seconds for `qpopper` and 0.047133 seconds for `sendmail`.

Giffin et al. [9] extended Wagner's work to application binaries for secure remote execution. They used null system call to eliminate impossible paths in their NFA model by placing a null system call after a function call. *Paid* is different from this work because it places a null system call only where non-determinism cannot be resolved through graph inlining and system call stub inlining. As a result, *Paid* can use the DFA model to implement a simple and efficient runtime verifier inside the kernel. Giffin et al. also tried graph inlining, which they called automaton inlining. They found graph inlining increases the state space dramatically, but *Paid*'s implementation on Linux only increases the state space only slightly. This discrepancy is due to the `libc` library on Solaris. For example, for a single `socket` call, it only needs a single edge or transition on

Linux, while it takes more than 100 edges on Solaris. They found numerous other library functions that share the same problem. Giffin’s PDA model is similar to Wagner’s model, and they used a bounded-stack to solve the infinite stack problem. However, when the stack is full, the PDA model eventually becomes a less precise NFA model. Giffin et al. also proposed a Dyck model [10] to solve non-determinism problem by placing a null system call before and after a function call to simulate stack operation. To reduce performance overhead, a null system call from a function call does not actually trap to the kernel if the function call itself does not make a system call.

Behavior blocking is a variation of system call-based intrusion detection. Behavior blocking systems run applications in a sandbox. All sandboxed applications can only have the privileges specified by the sandbox. Even if an application is hijacked, it cannot use more privileges than as specified. Existing behavior blocking systems include MAPbox [1], WindBox [3], Janus [11], and Consh [2]. The key issue of behavior blocking systems is to define an accurate sandboxing policy, which is

Systems such as StackGuard [6], StackShield [21] and RAD [5, 16] tried to protect the return addresses on the stack, which are common targets of buffer overflow attacks. Non-executable stack [19] prevents applications from executing any code on the stack. Another problem is that they cannot prevent attacks that target function pointers. IBM’s GCC extension [7] reorders local variables and places all pointer variables at lower addresses than buffers. This technique offers some protection against buffer overflow attacks, but not buffer underflow attacks. Purify [12] instruments binaries to check each memory access at run time. However, the performance degradation and the increased memory usage are the key issues that prevent Purify from being used in production mode. Kiriansky [14] checks every branch instruction to ensure that no illegal code can be executed.

3 Program Semantics-Aware Intrusion Detection

3.1 Overview

Paid includes a compiler that automatically derives a system call site flow graph or SCSFG from an application’s source code, and a DFA-based run-time verifier that checks the legitimacy of each system call. To be efficient, the run-time verifier of *Paid* is embedded in the kernel to avoid the context-switching overhead associated with user-level system call monitors. The in-kernel verifier has to be simple so that it itself does not introduce additional vulnerabilities. It also needs to be fast so as to reduce the performance overhead visible to applications. Finally it should not consume much of the kernel memory. The key challenge of *Paid* is to minimize the degree of non-determinism in the derived SCSFG such that the number of checks that the run-time verifier needs to perform is minimized.

Once an application’s SCSFG is known, the attacker can also use this information to mount a mimicry attack, in which the attacker follows a legitimate path through the application’s SCSFG until it reaches a system call she needs

to deal the fatal blow. For example, assume an application has a buffer overflow vulnerability and the system call sequence following the vulnerability point is `{open, setreuid, write, close, exec}`, and an attacker needs `setreuid` and `exec` for her attack. After the attacker hijacks the application's control using a buffer overflow attack, she can mimic the legitimate system call sequence by interleaving calls to `open`, `write` and `close` with those to `setreuid` and `exec` properly, thus successfully fooling any intrusion detection systems that check only system call ordering. To address mimicry attacks, *Paid* applies two simple techniques: stack integrity check and random insertion of null system calls. In the next version of *Paid*, we will add a comprehensive checking mechanism on system call arguments as well.

3.2 From NFA to DFA

The simplest way to construct a call graph for an application is to extract a local call graph for each function from the function's CFG, and then construct the final application call graph by linking per-function local call graphs using either *graph interlinking* or *graph inlining*, which are illustrated in Figure 1. A local call graph or an application call graph is naturally an NFA because of such control constructs as if-then-else and function call/return. To remove non-determinism, we employ the following techniques: 1) system call stub inlining, 2) graph inlining, and 3) insertion of `notify` system call.

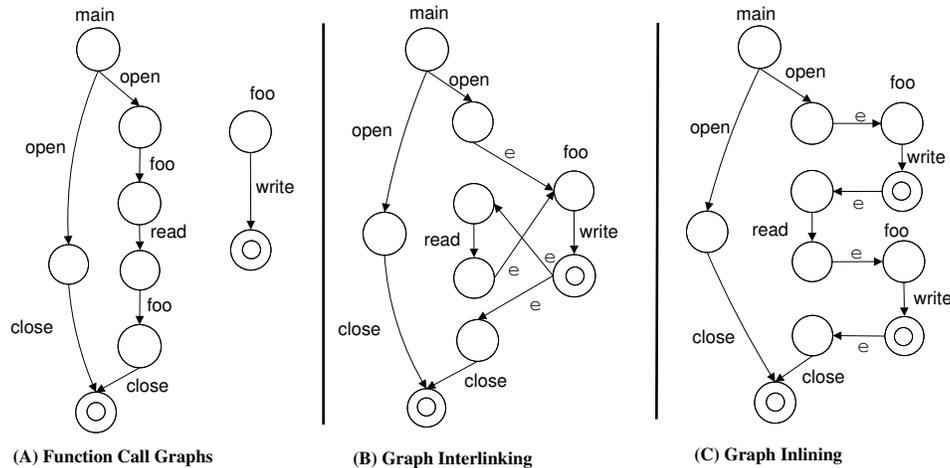


Fig. 1. Graph interlinking and graph inlining are two alternative to constructing a whole-program system call graph from the system call graphs of individual functions.

One source of non-determinism is due to functions that have many call sites. For these functions, the number of out-going edges of the final state of their local call graph is more than one, as exemplified by the function `foo` in Figure 1(B).

To eliminate this type of non-determinism, we use graph inlining as illustrated in Figure 1(C). In the application call graph, each call to a function with multiple call sites points to a unique duplicate of the function’s call graph, thus ensuring that the final state of each such duplicated call graph have a single out-going edge. Graph inlining can significantly increase the state space if not applied carefully. We use an ε -transition removal algorithm to remove all non-system call edges from a function’s CFG before merging the per-function call graphs.

Another source of non-determinism is due to control transfer constructs, such as `for` loop, `while` loop and `if-else-then`. One example of such problem is shown in Figure 1(C), where the program’s control can go to two different states from the first state, at which an `open` system call is made. The reason for this non-determinism is that on a Linux system, system calls are made indirectly through system call stubs. Therefore, it is not possible to differentiate the open system call made in the then branch from that in the else branch. We address this problem by uniquely identifying each system call so that when a system call is made, the runtime checker knows who it is. More concretely, we inline every system call with its associated system call stub so that a system call can be uniquely identified with its return address.

System call stubs inlining does not completely solve the non-determinism problem due to flow control constructs, since it does not inline normal functions. An example of such non-determinism is shown as follows:

```

a()          b()          c()          main()
{           {           {           {
  open();   {           {           {
}           a();       a();       if(true)
}           read();    close();    b();
}           }           }           else
}           }           }           c();
}           }           }           }

```

Since the functions `a`, `b`, and `c` are not inlined, the then branch and the else branch of `main` eventually lead to the same open system call even though the open system call stub is inlined. As another example, graph inlining cannot completely eliminate all non-determinism for recursive functions with multiple call sites, either. *Paid* introduces a new system call called `notify` to resolve whatever non-determinism is left after graph inlining and system call stub inlining are applied. The *Paid* compiler compiles an application in two passes. The first pass generates an NFA, and *Paid* visits every state of the NFA to detect non-determinism. When there is non-determinism, it traces back to the initial point that leads to the non-determinism and marks it accordingly. In the second pass the compiler inserts a `notify` call to each marked point to remove the corresponding non-determinism, and finally generates a DFA. Giffin et al [9, 10] used a similar idea to eliminate non-determinism due to multi-caller functions. However, blindly inserting `notify` calls can incur high performance overhead. *Paid* only inserts `notify` calls when non-determinism cannot be resolved by system call stub inlining and automaton inlining.

With system call stub inlining, graph inlining and `notify` system call insertion, the final call graph generated by the *Paid* compiler is actually a DFA, which we refer to as System Call Site Flow Graph or SCSFG. In addition to system call ordering, SCSFG also captures the exact location where each system call is made since all system call stubs are inlined. Also the final SCSFG does not

contain any non system call, which reduces the state space dramatically since most of the function calls do not contain any system calls.

3.3 System Call Inlining

In Linux, system calls are made through system call stubs, and the actual trap instruction that transfers control to the kernel is `int $0x80` in the Intel X86 architecture. *Paid* inlines each system call site with the associated stub and the address of the instruction following `int $0x80` becomes the unique label for the system call site. Figure 2(B) shows the result of inlining system call sites. LABEL_foo_0001 is the call site of the `write` system call in the *then* branch, and LABEL_foo_0003 is the call site of the `write` system call in the *else* branch. Figure 2(C) shows the SCFSG for the function `foo`, which includes a unique transition edge for each of the `write` system calls.

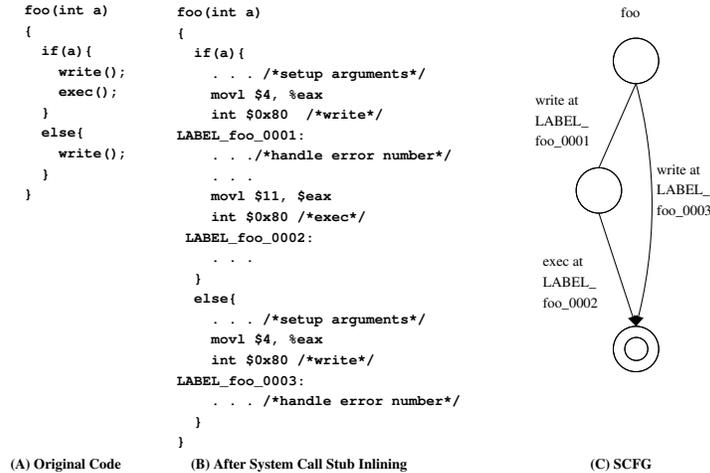


Fig. 2. After system call site inlining, the two `write` system calls can be distinguished based on their unique labels, as indicated by LABEL_foo_0001 and LABEL_foo_0003 in the result of inlining (B). Accordingly, the SCFSG in (C) has two different `write` transition edges.

System call site inlining is implemented via GCC front-end's function inlining mechanism. However, to exploit this mechanism, we need to rewrite all system call stubs so that they are suitable for inlining. Rewriting system call stubs turns out to be a time-consuming task because of various idiosyncrasies in the stubs. Some system call stubs such as `open`, `read`, and `write` are actually generated through scripts. Other system call stubs such as the `exec` family, the `socket` family, and `clone` need to be modified by hand so that they conform to the call stub convention used in GLIBC. Finally, the LIBIO library, which replaces the old standard I/O library in the new version of GLIBC, turns out to consume

most of the rewriting effort. Although *Paid* does not inline normal functions, it chose to inline `fopen` and `fwrite` because they are important to system security. However, because `fopen` and `fwrite` use other functions in the LIBIO library and the actual `open` and `write` system call is made through a function pointer, we are forced to modify the whole LIBIO library so that function pointers are eliminated and the resulting functions are suitable for inlining.

To force GCC automatically to inline a function, the function has to be declared as `always_inline`, and the function has to be parsed before its callers. Therefore, all rewritten system call stubs are declared as `always_inline`, and are put in a header file called `syscall_inline.h`. The only modification to GCC for system call stub inlining is to make sure that the `syscall_inlined.h` is always loaded and parsed first before other header files and the original source file.

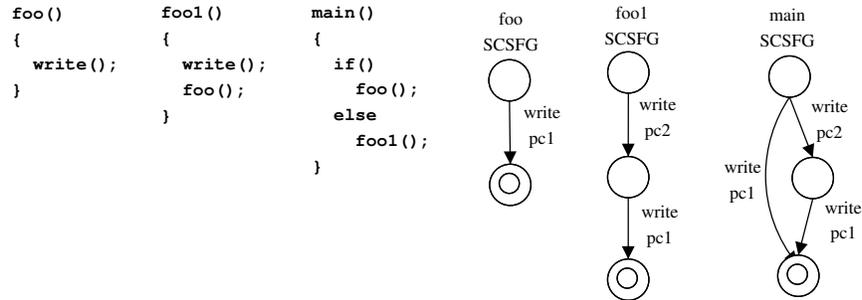


Fig. 3. *Paid's* compiler builds a SCSFG for each function, and the SCSFG for `main` represents the SCSFG for the entire program because of graphs inlining.

3.4 Building System Call Site Flow Graph

Paid needs two passes to compile an application. The two passes are very similar except that during the first pass, *Paid* analyzes the resulting SCSFGs to find out where to insert notify calls to eliminate non-determinism. The second pass builds the final SCSFGs into a DFA model. *Paid's* compiler generates a call site flow graph (CSFG) for each function, which contains normal function calls and system calls in the function. Then starting with `main's` CSFG, *Paid's* linker uses a depth first search algorithm to build a SCSFG for each function in a bottom up fashion. For example, if function `foo1` calls function `foo2`, `foo2's` SCSFG is constructed first, and is then duplicated and inlined in the SCSFG of `foo1`. If the linker detects a recursive call chain, it inlines every function in the call chain and assigns the resulting graph as the call graph of the first function in the chain. For example, for the recursive call chain (`a->b->c->a`), the linker inlines the CSFG of `c` and `b` at their respective call sites and assigns the expanded graph to `a`. The result of inlining turns a call chain into a self-recursive call. After a SCSFG is generated, the linker uses a simple ϵ -transition removal algorithm to walk through the whole graph and remove all non-system-call nodes. In the end,

each function's SCSFG contains only system call nodes, and an entry node and exit node. Because *Paid* inlines each function call graph, the SCSFG of `main` is the SCSFG for the whole program, which the run-time verifier uses to check each incoming system call. Finally, the linker stores all SCSFGs that contain at least one system call node, in an `.scsfg` section of the final ELF executable binary. Figure 3 shows a short program with its SCSFGs. It is still necessary to store the SCSFGs for individual functions because they may be needed when the main SCSFG is amended at run time, as described in the following subsection.

Many GLIBC functions are written in assembly language. We have to manually generate the CSFGs for those functions. The technique we use to generate CSFGs for functions written in assembly language is to compose dummy skeleton C functions that preserve the call sequence of these functions, and then use the *Paid* compiler to compile the skeleton C functions. The CSFGs for the skeleton C functions are by construction the CSFGs for the assembly-code functions.

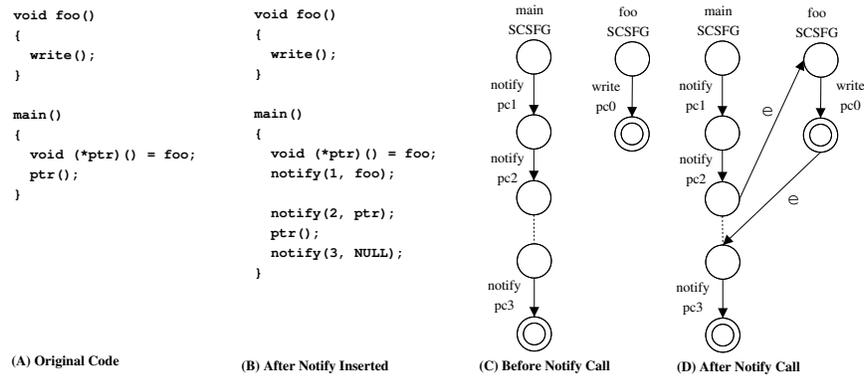


Fig. 4. This demonstrates how a `notify` system call associated with a function pointer amends an application's SCSFG at run time. (C) shows the SCSFGs before `notify` is called. (D) shows how `notify` links the main SCSFG and the SCSFG of the target function, `foo`.

3.5 Run-Time Notification

The original design of *Paid* assumes that the control flow graph of a program can be completely determined at compile time. This assumption is not valid because of branches whose target addresses cannot be determined statically, signals, unconventional control transfers such as `setjmps/longjmps`, etc. To solve this problem, *Paid* introduces a special run-time notification mechanism based on a special system call, `notify(int flag, unsigned address)`, which informs the run-time verifier of where the program control currently is at the time when the `notify` system call is made. *Paid* uses the `notify` system call for different purposes, each of which corresponds to a distinct value in the flag field. With

the help of `notify`, *Paid*'s run-time verifier can synchronize with the monitored application across control transfers that cannot be determined at compile time. Because `notify` is a system call and is thus also subject to the same sandboxing control as other system calls, attackers cannot issue arbitrary `notify` system calls and modify the victim application's system call pattern.

Instead of applying pointer analysis, *Paid* inserts a `notify` system call before every function call that uses a function pointer. The actual value of the function pointer variable, or the entry point of the target function, is used as an argument to the `notify` system call, as shown by the second `notify` call in Figure 4(B). When a `notify` system call that *Paid*'s compiler inserts because of a function pointer traps into the kernel, and the target function's SCSFG has not been linked to the main SCSFG at the current execution point, the run-time verifier searches for the SCSFG of the target function and dynamically links the matched SCSFG to the main SCSFG at the current execution point, as demonstrated by Figure 4(D). Because any function could potentially be the target of an indirect function call, *Paid* needs to store the SCSFGs for all functions. If a function is called from multiple call sites through a function pointer simultaneously, the main SCSFG may become an NFA. To avoid this, *Paid* also inserts a `notify` system call after every function call that uses a function pointer to inform the run-time verifier about which return path it should take. However, an attacker can overflow a function pointer to point to a desired existing function to exploit the attacks similar to return-to-libc attack. To reduce the attack possibilities, *Paid* also inserts a `notify` call at each location where a function's entry point is assigned to a function pointer, such as shown by the first `notify` call in Figure 4(B). The `notify` call informs the run-time verifier to mark the function, and the run-timer never links any unmarked function's SCSFG to the main SCSFG.

`setjmp` and `longjmp` functions are for non-local control transfers. To handle `setjmp/longjmp` calls correctly, the *Paid* compiler inserts a `notify` call before each `setjmp` call, using the address of the `jmp_buf` object as an argument. The `jmp_buf` data structure is modified to include a pointer of the `notify` call in the SCSFG. The added pointer is also used to pass the corresponding `setjmp` return address to a `notify` call. When a `notify` system call due to `setjmp` is made, the run-time verifier first retrieves the `setjmp` return address from the `jmp_buf` and stores it in the current `notify` node, and then the verifier stores the current location, which is the address of the current `notify` node, in the `jmp_buf` object. *Paid*'s compiler also inserts a `notify` system call before a `longjmp` call, using the address of the `jmp_buf` as an argument. Upon a `notify` call due to `longjmp`, the run-time verifier retrieves the previous location from the `jmp_buf` object and links this location to the current location if the saved `setjmp` return address matches the `longjmp` destination address stored in the `jmp_buf`.

Signals are handled differently. For non-blocking signals, the kernel either ignores the signal, executes the default handler or invokes the user-supplied signal handler. In the first two cases no user code is executed and so no modifications are needed. However, if the application provides a signal handler, say `handle_signal`, the run-time verifier first creates a new system call node for the `sigreturn` system call, which is pushed on the user stack by the kernel [], and links the new node to the final node of the SCSFG of `handle_signal`. Then the verifier saves the current SCSFG pointer in the new node, and changes the cur-

rent SCSFG pointer to the entry node of the SCSFG of `handle_signal`. Finally, the `handle_signal` is executed. When `handle_signal` returns back to the kernel through the `sigreturn` system call, the run-time verifier restores the current SCSFG pointer from the one saved in the node corresponding to the `sigreturn` system call, and proceeds as normal.

3.6 Stack Integrity Check

System call stub inlining and sandboxing based on system call sites/ordering constraints force an attack code to jump to the actual system call sites in the program in order to issue a system call. However, when control is transferred to a system call site, it is not always possible for the attack code to grab the control back. To further strengthen the protection, *Paid's* run-time verifier also checks the stack integrity by ensuring that all the return addresses in the stack are proper when a system call is made. If any return address is outside the original text segment, this indicates a control-hijacking attack may have occurred, because *Paid* explicitly forces all code regions to be read-only so that no attack code could be inserted into these regions. This simple stack integrity check greatly reduces the room that mimicry attacks have for maneuver as most such attacks need to make more than one system calls. Although gcc uses stack for function trampolines for nested functions, it does not affect the stack integrity check since function trampoline code does not make function calls. Also before executing a signal handler, Linux puts the `sigreturn` system call on the user stack and points the signal handler's return address to the `sigreturn` system call code in the stack. However, *Paid* can easily detect such return addresses by examining the code on the stack since the `sigreturn` invocation code is always the same.

3.7 Random Insertion of Null System Calls

To further improve the detection strength of *Paid*, *Paid* randomly inserts some null system calls into an application. A null system call is a `notify` system call that does not perform any operation. *Paid* randomly chooses some functions between two consecutive system calls in a function call sequence to insert random `notify` call. For example, for the call sequence `{write, buf, a, b, c, exec}`, *Paid* may insert a `notify` call in the function `a` so that the new call sequence would be `{write, buf, a, notify, b, c, exec}`. Assuming that a buffer overflow happens in the function `buf`, the attack code cannot call `exec` directly since `notify` is before `exec` in the sequence, and the `notify` call is also included in the SCSFG. If the attacking code wants to call `exec`, it has to setup the stack to regain the control after it makes a `notify` call. However, since the verifier checks the stack when the `notify` is made, it detects the illegal return address on the stack and terminate the process. To make the remote attacks more difficult, one may be willing to recompile server applications and hide the binaries from the remote attackers. Since `notify` calls are inserted randomly, it is highly unlikely that an attack code can guess their existence and their call sites. Inserting null system calls provides the run-time verifier more observation

points to monitor an application. Together with stack integrity check, it also forces attack code to follow more closely the application’s original control flow.

To randomly insert null system call, the first compilation pass analyzes the initial version of the SCSFGs and determines where to insert null system calls, and the second pass builds the final SCSFGs with random `notify` calls inserted. The exact algorithm for inserting null system calls in the current *Paid* prototype works as follows. For each two system calls A and B in an application SCSFG, if there exist a path from A to B where there is no other system call, *Paid* randomly chooses some functions on the path to insert null system call. If the number of functions on a path is 2 to 4, the compiler randomly inserts a null system call into one of the functions. If the number of function calls is 5 to 7, it randomly inserts 2 null system calls; if the number is 8 and up, it inserts 3. Empirical measurements show that the number of function calls between two consecutive system calls is mostly between 2 and 5, and is rarely more than 10. Many paths may go through a same function; however, *Paid* never inserts more than one null system calls in a function, and it always make sure that a null system call is inserted outside any loop.

Potentially an attacker can examine the binary to deduce the locations of these randomly inserted `notify` system calls. In practice, such attacks are unlikely because it is well known that perfect disassembly is not impossible on X86 architecture [16], due to the fact that distinguishing data and code is fundamentally undecidable.

3.8 Run-time Verifier

Linux’s binary loader is modified to load the `.scsfg` segment of an ELF binary into a randomly chosen region of its address space. That is, although the verifier performs system call-based sandboxing inside the kernel, the SCSFGs used in sandboxing reside in the user space, rather than in the kernel address space. To prevent attackers from accessing SCSFGs, the region at which the SCSFGs are stored is chosen randomly at load time. In addition, the SCSFG region is marked as read-only, so that it is not possible for an attacker to modify them without making system calls, which will be rejected because *Paid* checks every system call. When the run-time verifier needs to amend the main SCSFG due to a `notify` system call, it has to make the SCSFG region writable, and turns it back to read-only after the amendment. However, these operations are performed inside the kernel, and thus may override the region’s read-only marking.

An *execution pointer* is added into Linux’s `task_struct` to keep track of a process’s current progress within its associated SCSFG. For each incoming system call, the verifier compares the system call number and the call site with each child of the node pointed to by the execution pointer. If a match is found, the execution pointer is moved to the matched node; otherwise, it indicates that an illegal system call is made, and the verifier simply terminates the process. Because a program’s main SCSFG is a DFA, only a single execution pointer is needed and the SCSFG graph traversal implementation is extremely simple, only 45 lines of C code. Accordingly, the performance overhead of the run-time verifier is very small, as demonstrated in Section 5.

3.9 Support for Dynamically Linked Libraries

Paid treats the dynamically linked libraries as static linked libraries. It statically builds the SCSFGs for each DLL used by an application, and inlines the DLL SCSFGs to the main SCSFG of the application at the static linking time. However, for each DLL, there is a table to hold all call site addresses in the `.scsfg` segment. If a DLL is loaded at a different address than the preferred address, the loader will fix all its call site addresses in the associated table at load time. For libraries loaded by `dlopen` at run time, the pre-built SCSFGs of these libraries are copied to the application's own address space after the libraries are loaded. All library function calls via function pointers that are obtained by `dlsym` also rely on `notify` system calls, with an additional burden to fix call site addresses if a library is not loaded in the preferred location. The disadvantage of statically inlining the SCSFGs of the DLLs is that the resulting binaries cannot work with different versions of the same libraries.

3.10 Support for Threads

To support a multi-threaded application, *Paid* needs to maintain an execution pointer for each thread, and applies a mechanism similar to the way `setjmp/longjmp` is handled to switch the execution pointer at run time. At this point *Paid* does not support applications using user-level threads. However, this is not a limitation because most multi-threaded applications use kernel-level threads, which Linux supports through the `clone` system call. Although many applications use `pthread`, the `pthread` library of GLIBC actually uses kernel-level threads as well on Linux. We modify the `clone` and `fork` system calls so that the SCSFG region is copied when they are called.

4 Attack Analysis

Even though *Paid* uses several techniques to reduce the feasibility of mimicry attacks, there are two cases in which mimicry attacks are still possible as shown in Figure 5. First, if an attack code can compromise an application and then directly issue a damaging system call without requiring getting the control back, for example, when an `exec` system call immediately follows a buffer overflow vulnerability, *Paid* cannot stop this type of attacks. Second, if an attacker can mount an attack without making system calls explicitly, for example, by just manipulating the system call arguments of one or multiple legitimate system calls already in the program, *Paid* cannot stop this type of attacks, either. However, to the best of our knowledge these two types of attacks are very rare in practice.

Stack integrity check helps *Paid* to ensure that the stack state at the time a system call is made is proper. However, there is no guarantee that the monitored application indeed goes through the function call sequence as reflected in the chain of return addresses, because the attacker can easily doctor the stack frames before making any system call just to fool the stack integrity checking mechanism. None the less, stack integrity check at least greatly complicates the attack code, while incurring a relatively modest performance overhead, as shown in Section 6.

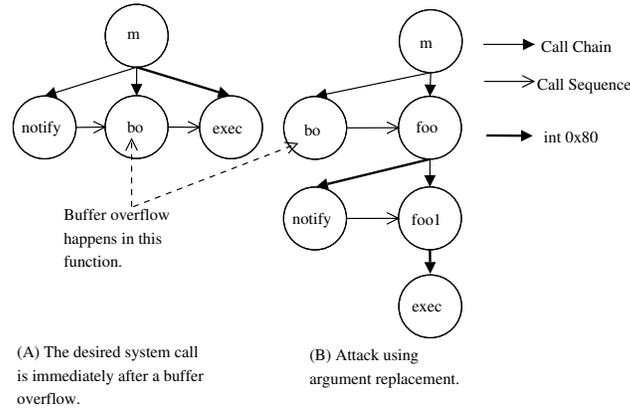


Fig. 5. This figure shows the two cases that *Paid* cannot handle. Figure (A) shows an attack that only needs a single system call and the desired system call is immediately after a buffer overflow. Figure (B) shows the argument replacement attack assuming that the arguments needed by `exec` are directly passed from the function `m` to the system call `exec` through the call chain. After the buffer overflow, the injected code sets up the desired arguments for the `foo` function, and then calls `foo` directly.

Because *Paid* embeds an application’s SCSFG inside its binary file, and checks the application’s run-time behavior against it, some ill-formed SCSFGs may cause *Paid*’s run-time verifier to misbehave, such as entering an infinite loop, thus leading to denial-of-service attacks. To the best of our knowledge, *Paid*’s run-time verifier does not have such weaknesses. As a second line of defense, *Paid* can perform run-time checks only for trusted applications, in a way similar to how only certain applications have their `setuid` bit turned on. Finally, the run-time verifier can include some self-monitoring capability, so that it can detect anomalous system call graphs and terminate the process.

5 Performance Evaluation

5.1 Prototype and Methodology

The current *Paid* compiler is derived from GCC 3.1 and GNU ld 2.11.94 (linker), and runs on Red Hat Linux 7.2. The *Paid* prototype can successfully compile the whole GLIBC (version 2.2.5), and production-mode network server programs implemented using `fork` or `clone` system call, such as Apache and Wu-ftp. For this study, we used as test programs the set of network server applications shown in Table 1, and compared *Paid*’s performance and space requirement with those of GCC 3.1 and Red Hat Linux 7.2, which represent the baseline case. To analyze detailed performance overhead, we conducted each experiment in four different configurations: plain *Paid* that only uses SCSFGs (plain *Paid*), *Paid* with stack integrity check (*Paid/stack*), *Paid* with random insertion of null system calls (*Paid/random*), and *Paid* with both stack integrity check and random null system calls (*Paid/stack/random*).

To test the performance of each server program, we used two client machines to continuously send 2000 requests to the tested server program. In addition, we modified the server machine's kernel to record the creation and termination time of each process. The throughput of a network server application is calculated by dividing 2000 by the time interval between the creation of the first forked process and the termination of the last forked process. The latency is calculated by taking the average of the run time used by the 2000 forked processes. The Apache web server program is handled separately in this study. We configured Apache to handle each incoming request with a single child process so that we could accurately measure the latency of each Web request. However, the throughput of Apache decreases significantly as a result of this set-up.

The server machine is a 1.5-GHz P4 with 256MB memory, one client machine is a 300-MHz P2 with 128MB memory and the other client is a 1.1-GHz P3 with 512MB memory. They are connected through an isolated 100Mbps Ethernet link. All machines run Redhat Linux 7.2. To test http and ftp servers, the client machines continuously fetched a 1-KByte file from the server, and the two client programs were started simultaneously. In the case of pop3 server, the clients checked mails and retrieved a 1-KByte mail from the server. For the sendmail server, two clients continuously send a 1-KByte mail to two different users. All client programs used in the test were modified in such a way that they continuously send 2000 requests to the server. A new request was sent only after the previous one is completely finished. To speed up the request sending process, client programs simply discarded the data returned from the server. All network server programs tested were statically linked, and the modified GLIBC-2.2.5 library was recompiled by the *Paid* compiler.

5.2 Effectiveness

Two small programs with buffer overflow vulnerability were used to test the effectiveness of *Paid* in stopping attackers from making unauthorized system calls. The first program allows attackers to overflow the return address of one of its functions and point it to a piece of injected code that makes malicious system calls. *Paid's* run-time checker successfully stopped the execution of the program since the injected system calls were not the next valid system calls in the original program's main SCSFG. The second program allows attackers to overflow a function pointer to point to a piece of dynamically injected code. Existing buffer overflow defense systems such as Stackguard/Stackshield and RAD cannot handle this type of attacks. However, the notify system call in *Paid* successfully detects this attack because the SCSFG of the injected code was not found in the original program's `.scsfg` segment. We also tested *Paid* on the "double free" vulnerability of wu-ftp-2.6.0. The exploit script is based on the program written by TESO Security [20]. Again *Paid* is able to successfully stop this attack while the exploit successfully spawns a root shell from the ftpd that is compiled by the original GCC compiler.

5.3 Performance Overhead

Paid adds an extra `.scsfg` segment to an application's binary image to store all SCSFGs. The size of the `.scsfg` segment is expected to be large because it

Table 1. Characteristics of a set of popular network applications that are known to have buffer overflow vulnerability. The source code line count includes all the libraries used in the programs, excluding `libc`.

Program Name	Lines of Code	Brief Description
Qpopper-4.0	32104	Pop3 server
Apache-1.3.20	51974	Web server
Sendmail-8.11.3	73612	Mail server
Wu-ftpd-2.6.0	28055	Ftp server
Proftpd-1.2.8	58620	Ftp server
Pure-ftpd-1.0.14	28182	Ftp server

Table 2. The binary image size and compilation time overhead of *Paid* compared with *GCC*. The absolute size of the `.scsfsg` segment of each network application in bytes is also listed.

Program Name	Plain <i>Paid</i> Binary Size Overhead	Plain <i>Paid</i> SCSFGs Size	Plain <i>Paid</i> Compile Time Overhead	Random <i>Paid</i> Binary size Overhead	Random <i>Paid</i> SCSFGs Size	Random <i>Paid</i> Compile Time Overhead
Qpopper-4.0	124.86%	798,780	119.30%	155.05%	996,840	121.00%
Apache-1.3.20	156.38%	1,338,739	232.98%	177.99%	1,529,224	247.92%
Sendmail-8.11.3	153.87%	1,845,208	198.15%	184.79%	2,073,524	211.53%
Wu-ftpd-2.6.0	149.47%	1,078,448	164.73%	173.04%	1,253,412	197.45%
Proftpd-1.2.8	178.02%	1,590,262	169.84%	194.12%	1,738,270	194.18%
pure-ftpd-1.0.14	116.27%	680,820	100.12%	128.85%	755,304	120.85%

contains the SCSFGs of all the functions in the application as well as in the libraries that the application links to. Because of the `.scsfsg` segment, and because of system call inlining, the binary image of a network application compiled under the *Paid* compiler is much larger than that compiled under *GCC*. The binary image space overhead of the test applications ranges from 116.27% to 178.02% for the applications compiled by plain *Paid*, and from 128.85% to 194.12% for the applications compiled by *Paid* with random null system calls, as shown in Table 2. Most of the space overhead is indeed due to the `.scsfsg` segment. The absolute size of this segment for each of the test network applications is also shown in Table 2. Note that this increase in binary size only stresses the user address space, but has no effect on the kernel address space size.

The *Paid* compiler also needs more time to extract the SCSFGs. Table 2 shows that the additional compilation time overhead of *Paid* when compared with *GCC* is from 100.12% to 232.98% for plain *Paid*, and from 120.85% to 247.92% for *Paid*/random. Under *Paid*'s compiler, compilation of applications takes two passes. This compilation time overhead is mainly due to the second pass.

The performance overhead of *Paid* mainly comes from the additional check at each system call invocation. More specifically, it involves stack integrity check and the decision logic required to move to next DFA state. We measured the average latency penalty at each system call due to this check, and the results in Table 5 show that this penalty is between 5.43% to 7.48%. However, the overall latency penalty of plain *Paid* compared to the base case (*GCC* + generic Linux) is smaller, as shown in Table 3 and Table 4, because each network application

Table 3. The latency penalty of each network application compiled under *Paid* with different configurations when compared with the baseline case.

Program	paid Latency Penalty	paid/stack Latency Penalty	paid/random Latency Penalty	paid/stack/rand Latency Penalty
Qpopper-4.0	5.69%	5.84%	6.42%	6.63%
Apache-1.3.20	5.14%	5.70%	6.93%	7.63%
Sendmail-8.11.3	7.31%	8.38%	10.32%	11.66%
Wu-ftpd-2.6.0	2.28%	2.76%	3.73%	4.58%
Proftpd-1.2.8	6.85%	7.63%	8.55%	9.85%
pure-ftpd-1.0.14	4.80%	5.33%	5.10%	7.58%

also spends a significant portion of its run time in the user space. As a result, the overall latency penalty of plain *Paid* ranges from 2.28% (wu-ftpd) to 7.31% (sendmail), and the throughput penalty ranges from 2.23% (Wu-ftpd) to 6.81% (Sendmail). These results demonstrate that despite the fact that *Paid* constructs a detailed per-application behavior model and checks it at run time, its run-time performance cost is really modest. This is also true for the *Paid/stack/random* configuration, which includes both stack check and random insertion of null system calls. The latency penalty for *Paid/stack/random* is from 4.58% (Wu-ftpd) to 11.66%(Sendmail), and the throughput latency penalty ranges from 4.38% (Wu-ftpd) to 10.44% (Sendmail).

Compared with plain *Paid*, *Paid/Stack* only increases the performance overhead slightly as shown in Table 3 and Table 4. The latency penalty of *Paid/Stack* ranges from 2.76% to 8.38%, and the throughput penalty ranges from 2.69% to 7.73%. This shows that checking the stack integrity at every system call is a relatively inexpensive verification mechanism. In contrast, *Paid/random* incurs more overhead than *Paid/stack*, because each null system call inserted incurs expensive context switching overhead. As the number of null system calls inserted increases, this overhead also increases, but the strength of protection against mimicry attacks also improves as attack codes are forced to follow more closely the application's original execution flow. The latency penalty for *Paid/random* ranges from 3.73% to 10.32%, and the throughput penalty ranges from 3.60% to 9.36%.

One of the major concerns early in the development cycle of this project is the performance overhead associated with `notify` system calls, which are used to amend the main SCSFG at run time when functions are called through function pointers or other dynamic control transfers that cannot be determined at compile time. Amending a SCSFG may take a non-negligible amount of time. In addition, the SCSFG region and the SCSFG's current execution point need to be copied to the child process as part of `fork` or `clone`. Copying of the SCSFG region actually does not incur serious performance overhead because Linux uses copy-on-write. However, when `notify` system calls are made, the SCSFG needs to be modified, and additional data copying is required to implement copy-on-write.

To study the impact of `notify` system calls due to function pointers on the application performance, we instrumented the *Paid* compiler and the run-time

Table 4. *The throughput penalty of each network application compiled under Paid with different configurations when compared with the baseline case.*

Program	paid Throughput Penalty	paid/stack Throughput Penalty	paid/random Throughput Penalty	paid/stack/rand Throughput Penalty
Qpopper-4.0	5.38%	5.52%	6.03%	6.22%
Apache-1.3.20	4.89%	5.39%	6.48%	7.09%
Sendmail-8.11.3	6.81%	7.73%	9.36%	10.44%
Wu-ftp-2.6.0	2.23%	2.69%	3.60%	4.38%
Proftpd-1.2.8	6.41%	7.10%	7.87%	8.96%
pure-ftp-1.0.14	4.58%	5.06%	4.85%	7.05%

verifier to measure the `notify` system call frequency. Table 5 lists the number of `notify` calls inserted into each tested network application and the modified GLIBC statically. We also collected the actual number of `notify` calls made at run time. The number of dynamic `notify` calls includes both calls from the applications and the modified GLIBC. If the SCSFG of a function has been linked to the main SCSFG by a `notify` call at the same call site previously, the verifier does not need to search for the SCSFG again. The last column of the table shows the number of `notify` calls that actually need a full-scale search through the SCSFGs, or the number of `notify` cache misses.

The number of statically inserted `notify` calls ranges from 0 (`wu-ftp`) to 187 (`Apache`), and the number of actual calls ranges from 92 (`pure-ftp`) to 981 (`Apache`). The number of `notify` calls that need full SCSFG search ranges from 7 (`wu-ftp`) to 33 (`Proftpd`). Even though `Apache` makes 981 `notify` calls at run time, its latency penalty is 7.63% and the throughput penalty is 7.09%. This low overhead mainly comes from the surprising low overhead associated with `notify` system calls. When SCSFG search is not needed, a `notify` system call only needs 1,745 CPU cycles. The average time requirement for a `notify` system call that needs SCSFG search is 3,383 CPU cycles. However, most of `notify` system calls do not need SCSFG search.

Table 5. *The average per-system call latency penalty, the number of static and dynamic notify system calls, and the number of dynamic notify system calls that need full-scale SCSFG search.*

Program Name	Average System Call Overhead	Static Notify Count	Dynamic Notify Count	Notify Calls Need to Search for SCSFG
Qpopper-4.0	7.48%	65	49	7
Apache-1.3.20	5.61%	187	981	10
Sendmail-8.11.3	7.06%	39	386	16
Wu-ftp-2.6.0	6.09%	0	99	7
Proftpd-1.2.8	6.89%	133	673	33
Pure-ftp-1.0.14	5.43%	4	92	23
modified-GLIBC	N/A	154	N/A	N/A

6 Conclusion

System call-based intrusion detection provides the last line of defense against control-hijacking attacks because it limits what attackers can do even after they successfully compromise a victim application. The Achilles' heel of system call-based intrusion detection is how to efficiently and accurately derive a system call model that can be tailored to individual network applications. This paper describes the design, implementation, and evaluation of *Paid*, a fully operational compiler-based system call-based intrusion detection system that can automatically derive a highly accurate system call model from the source code of an arbitrary network application. One key feature of *Paid* is its ability to exploit run-time information to minimize the degree of non-determinism that is inherent in a pure static analysis approach to extracting system call graph. The other unique feature that sets *Paid* apart from all existing system call-based intrusion detection systems including commercial behavior blocking products, is its application of several techniques that together reduce the vulnerability window to mimicry attacks to a very small set of unlikely program patterns. As a result, we believe *Paid* represents one of the most comprehensive, robust, and precise host-based intrusion detection systems that are truly usable, scalable, and extensible. Performance measurements on a fully working *Paid* prototype show that the run-time latency and throughput penalty of *Paid* are under 11.66% and 10.44%, respectively, for a set of popular network applications including the Apache web server, the Sendmail SMTP server, a Pop3 server, the wu-ftpd FTP daemon, etc. Furthermore, by using a system call inlining technique, *Paid* dramatically reduces the run-time system call overhead. This excellent performance improvement mainly comes from the fact that the SCSFGs that *Paid's* compiler generates is a DFA.

Currently, we are working on extending the *Paid* prototype in the following directions. First, we are developing compiler techniques that can capture system call arguments whose values can be statically determined or remain fixed after initialization. Being able to check system call arguments further shrinks the window of vulnerability to control-hijacking attacks. Second, we are exploring the feasibility of applying the same security policy extraction methodology on binary programs directly, so that even legacy applications whose source code is not available can enjoy the protection that *Paid* can provide.

References

1. A. Acharya and R. Mandar. Mapbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the Tenth USENIX Security Symposium*, 2000.
2. A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. In *USENIX Ann. Technical Conf*, 99.
3. D. Balfanz and D. R. Simon. Windowbox: a simple security model for the connected desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 37–48, 2000.
4. CERT Corrdination Center. Cert summary cs-2003-01. <http://www.cert.org/summaries/>, 2003.

5. T. cker Chiueh and F.-H. Hsu. Rad: A compiler time solution to buffer overflow attacks. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, April 2001.
6. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.
7. H. Etho. Gcc extension for protecting applications from stack-smashing attacks. <http://www.trl.ibm.com/projects/security/ssp/>.
8. H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 62–76, Berkeley, CA, May 2003. IEE Press.
9. J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. *USENIX Security Symposium*, August 2002.
10. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. *11th Annual Network and Distributed System Security Symposium*, February 2004.
11. I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.
12. R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, 1992.
13. S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.
14. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, August 2002.
15. N. Nguyen, P. Reiher, and G. H. Kuenning. Detecting insider threats by monitoring system call activity. In *IEEE Information Assurance Workshop*, United States Military Academy West Point, New York, June 2003.
16. M. Prasad and T. cker Chiueh. A binary rewriting approach to stack-based buffer overflow attacks. In *Proceedings of 2003 USENIX Conference*, June 2003.
17. V. Prevelakis and D. Spinellis. Sandboxing applications. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 119–126, Berkeley, CA, June 2001. USENIX Association.
18. R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.
19. Solar Designer. Non-executable user stack. <http://www.false.com/security/linux-stack/>.
20. TESO Security. x86/linux wu_ftp_d remote root exploit. <http://packetstormsecurity.nl/0205-exploits/7350wurm.c>.
21. Vendicator. Stackshield: A “stack smashing” technique protection tool for linux. <http://www.angelfire.com/sk/stackshield/>.
22. D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, Oakland, CA, May 2001. IEEE Press.
23. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.
24. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models, May 1999.