

Application Specific Sandboxing for Win32/Intel Binaries

Wei Li Lap-chung Lam Tzi-cker Chiueh
Computer Science Department
Stony Brook University

Abstract

Comparing the system call sequence of a network application against a sandboxing policy is a popular approach to detecting control-hijacking attack, in which the attacker exploits such software vulnerabilities as buffer overflow to take over the control of a victim application and possibly the underlying machine. The long-standing technical barrier to the acceptance of this system call monitoring approach is how to derive accurate sandboxing policies for Windows applications whose source code is unavailable. In fact, many commercial computer security companies take advantage of this fact and fashion a business model in which their users have to pay a subscription fee to receive periodic updates on the application sandboxing policies, much like anti-virus signatures. This paper describes the design, implementation and evaluation of a sandboxing system called *BASS* that can automatically extract a highly accurate application-specific sandboxing policy from a Win32/X86 binary, and enforce the extracted policy at run time with low performance overhead. *BASS* is built on a binary interpretation and analysis infrastructure called BIRD, which can handle application binaries with dynamically linked libraries, exception handlers and multithreading, and has been shown to work correctly for a large number of commercially distributed Windows-based network applications, including IIS and Apache. The throughput and latency penalty of *BASS* for all the applications we have tested except one is under 8

1 Introduction

One popular approach to host-based intrusion detection is to compare the run-time system call behavior of an application program with a pre-defined system call model, and declare an intrusion when a deviation between the two arises. This approach has been the linchpin of many research prototypes and commercial products under the name of sandboxing [20], behavioral blocking [7], restricted execution environment [12], etc. While conceptually appealing, the

technology has not been widely adopted in practice because the number of false positives, which disrupt legitimate applications, is still too high to be acceptable. Therefore, the main technical barrier of this system call-based sandboxing approach is how to automatically generate a system call model (or sandboxing policy) for arbitrary application programs that minimizes both the false positive rate and false negative rate. This paper describes the design, implementation and evaluation of a system call-based sandboxing system called *BASS* that successfully removes this barrier for commercially distributed Win32 binaries running on Intel X86 architecture.

BASS's automated system call model extraction mechanism is an extension of *PAID* [16], which analyzes an input program's source code and outputs a system call graph that specifies the ordering among the program's system calls. *BASS* extends *PAID* in several important ways. First, *BASS*'s system call model records the "coordinate" of each system call site, which is defined by the sequence of function calls from the program's main function to the function containing the system call site and the system call site itself [2]. Moreover, the run-time system call monitoring engine of *BASS* features a novel system call graph traversal algorithm that can efficiently map out the trajectory from one system call site to the next based on their coordinates. Second, *BASS* checks system call arguments in addition to system call ordering and coordinates. Finally, *BASS* supports load-time random insertion of null system calls to thwart mimicry attacks (explained later). As a result of these techniques, the false positive rate of *BASS* with is zero, i.e., whatever intrusions *PAID* reports are guaranteed to be an intrusion. In addition, the false negative rate of *BASS* with respect to control-hijacking attacks is very small, i.e., the probability of successful control-hijacking attacks is miniscule, as explained later in the Attack Analysis section.

Another major difference between *BASS* and *PAID* is *BASS* is able to derive a system call model for an arbitrary Windows/X86 executable file and dynamically linked library (DLL). Because state-of-the-art disassemblers cannot distinguish between instructions and data in Windows/X86 binaries with 100% accuracy [21], it is not possible to stati-

cally uncover all instructions of a binary image, let alone its system call model. To solve this problem, *BASS* is built on a general binary analysis and instrumentation infrastructure called *BIRD* [18], which is specifically designed to facilitate the development of software security systems by simplifying the analysis and instrumentation of Windows/X86 binaries. Given a binary program, *BIRD* statically disassembles it to uncover as many instructions as possible, rewrites it to allow run-time interception at all indirect jumps and calls, and dynamically disassembles those binary areas that cannot be disassembled statically.

The Windows operating environment also introduces several additional issues that do not exist in *PAID*, which was designed for the Linux platform. First, Windows binaries are more difficult to disassemble than Linux binaries, because the former tend to contain more hand-crafted assembly instruction sequences that violate standard programming conventions, such as jumping from one function into the middle of another function. Second, because the procedural call convention is not strictly followed, deriving the coordinate of a system call site is non-trivial as it is not always possible to accurately infer the locations of the return addresses currently on the stack. Third, Windows applications use DLLs extensively, and common DLLs such as `Kernel32.DLL`, `User32.DLL` and `NTDLL.DLL` are enormous. So it is essential to share the system call graphs for these DLLs across applications as well as their code. *BASS* successfully solves all these three problems, and demonstrates for the first time that it not only is feasible but also can be quite efficient to sandbox Windows binaries with an automatically generated system call model that produces zero false positive and close-to-zero false negatives. As a result, we believe *BASS* makes a powerful building block for guarding enterprises against all internet worms that use control-hijacking attacks such as buffer overflow attacks.

2 Related Work

2.1 Binary Analysis and Instrumentation

To construct a system call model from a binary, we need to reconstruct the control flow graph (CFG) from the binary by analyzing and disassembling the binary code. The implementations of Giffin and Feng et al. [10, 8] relied on the EEL library [17] to reconstruct the control flow graphs (CFG) from the binaries. The EEL library is designed to be a system-independent binary editing tool for analyzing and modifying executable programs. EEL depends on the symbol table of a binary to get the starting addresses of its procedures. If the symbol table was not available, EEL employs simple static disassembling techniques to discover

the procedure entry points. EEL was implemented for the SPARC architecture, whose instruction set is much simpler than the X86 architecture. Therefore, EEL's simple disassembling techniques are not powerful enough to discover procedure entry points for applications running on the X86 architecture.

OM [25] is a link-time binary optimization tool, which disassembles an input binary into an intermediate form or a generic register-transfer language. Application developers then can use the intermediate form to optimize their applications by modifying the intermediate form. Finally, OM translates the modified intermediate form into the target binary format. OM relies on the relocations tables, which are available at the link time. However, OM may not work on the binaries that do not have a symbol table. OM is further enhanced and evolves into *ATOM* [24], which provides a general framework for building customized program analysis tools. Both OM and *ATOM* only run on a RISC architecture, whose instruction set is less complex than the x86 architecture, the target of *BASS*.

Vulcan [23] (a binary transformation infrastructure) and *Diablo* [5] (a link-time rewriting framework) are designed to work with X86 binaries. However, *Vulcan* requires information from PDB files associated with binaries. The PDB file is generated by Microsoft Visual C++ using a specific compiler option and includes procedure name, symbol table, variable name/type information, etc. *Diablo* only works with the GCC-based tool chain. Otherwise, it needs to patch the tool chain to preserve some code and data information. Neither of them can operate commercially distributed Win32/X86 binaries. *Etch* [22] is an instrumentation and optimization framework that can work on Win32/Intel executables. The only paper on *Etch* [22] identified the challenges for Win32/Intel binary rewriting without providing any concrete solutions.

Dynamo [3] is a binary interpretation and optimization system running on HP PA-8000 machines under HP-UX 10.20 operating system. Its key idea is to use a software-based architectural emulator to detect so-called hot traces, i.e. sequences of frequently executed instructions, and optimize them dynamically so that they can run faster. *Dynamo* has been ported to the Win32/x86 platform [4]. It turns out that the Win32/x86 version runs much slower and incurs an overhead of about 30% to 40%. The reasons behind this are lack of documentation on Win32 API and additional implementation complexities that are not present on UNIX platform. Like *BIRD*, *Dynamo* can serve as a foundation for security applications. Program shepherding [13] is one such example. Compared with *Dynamo*, *BIRD* uses a disassembler rather than a software-based architectural emulator to interpret instructions, and thus significantly reduces the implementation complexity.

2.2 System Call-based Sandboxing

Wagner and Dean [28] proposed to use static analysis to extract directly from an application’s source code its system call model. They developed and compared three system call models: callgraph model, abstract stack model, and digraph model. The callgraph model is essentially a non-deterministic finite state automaton (NFA) model since it is generated directly from the control flow graph (CFG), and cannot resolve non-determinism due to conditional branches and multiple call sites to the same function. Such non-determinism provides more opportunities for attackers to exploit a class of attacks call mimicry [29] attacks. Therefore, they proposed a more expensive model called abstract stack model or non-deterministic pushdown automation model (NPDA) to remove the non-determinism dues to multiple call sites. Since the NPDA incurs too much runtime and space overhead, they proposed a less accurate but more efficient model called digraph model, which is similar to the system call sequence model proposed by Forrest et al [9]. Giffin et al. [10] extended the NFA and the NPDA models to binaries, and improved the efficiency and accuracy by using some optimization methods such as null system call insertion.

To further remove non-determinism, Giffin et al [11] also proposed a Dyck model, which inserts null system calls before and after a function call in order to retrieve the application context information. However, the Dyck model still contains non-determinism in the case of recursive functions, and the performance of the Dyck model is unpredictable because the considerable number of inserted null system calls. The *PAID* system developed by Lam and Chieh [16] employs a different approach to remove the non-determinism totally from their SCSFG model. *PAID* uses graph inlining and system call stub inlining to remove the non-determinism due multiple call sites, and it uses null system call insertion to remove the non-determinism due to control constructs. Compared with the above models, the SCSFG model is the most accurate and efficient model since it can use a deterministic finite state automaton or DFA algorithm to implement the graph traversal algorithm. Although the SCSFG model is a deterministic model, it requires substantial modification to the IO library and system call stubs, which make it more difficult to port it to a new LIBC. It also requires static linking to analyze where to insert null system calls.

The VPStatic/DPDA model proposed by Feng et al. [8] is the closest to *BASS*. Both the VPStatic model and the *BASS* model use return addresses to identify each call site and to remove the non-determinism due to multiple call sites. However, the VPStatic model does not remove non-determinism due to functions that contain a system call em-

bedded in an if-else-then construct, and to functions that are called in a loop. The fact that it does not take into account the return address of the trap instruction used in system calls also makes it vulnerable to mimicry attacks. In contrast, *BASS* removes all non-determinism in the programs through a novel system call graph traversal algorithm, and it can operate on Windows binaries directly.

We believe *BASS* represents one of the most comprehensive and efficient host-based intrusion detection system against control hijacking attacks and on the Win32/X86 platform. It is able to handle all production-mode Windows binaries that we have tested so far, including the MS Office suite, IIS, and IE, as well as well-known third-party binaries including Acrobat Reader, Apache, and FTP daemon. As for completeness, *BASS* supports system call monitoring for dynamically linked libraries, multi-threading, and exception handlers.

3 Application-Specific Sandboxing

3.1 Abstract Model

By preventing applications from issuing system calls in ways not specified in their system call model, one could effectively stop all control-hijacking attacks. One way to automatically derive a network application’s system call model is to extract its system call graph from its control flow graph (CFG) by abstracting away everything except the function call and system call nodes. A system call graph is a non-deterministic finite state automaton (NFA) model, due to if-then-else statements and functions with multiple call sites. For example, in Figure 1, because of an if-then-else statement the control can move to either `call_r1` or `call_r2` after `getuid_r0.t1` is called. The path $\{call_r1 \rightarrow Entry(n) \rightarrow Exit(n) \rightarrow ret_r2\}$ represents an impossible path [28], which cannot occur in the original program’s execution, but is allowed by the model. The more impossible paths exist in a system call model, the more leeway is made available to mimicry attacks [29], which issues system calls exactly in the same order as specified in the system call graph before reaching the system call that can damage the victim system (e.g., `exec()`).

To reduce the amount of non-determinism in a system call graph, *BASS* uses a Call Site Flow Graph (CSFG), which captures both the ordering among system call sites and their exact locations. More specifically, a system call site’s *coordinate* is uniquely identified by the sequence of return addresses on the user stack when it is made and the return address of the system call’s corresponding trap instruction. As shown in Figure 1, each system call node in a CSFG is labeled by the return address of its call

stub and the return address of its trap instruction, such as `getuid_r0.t1`. In this case, `r0` is the return address of the system call stub `getuid`, and `t1` is the return address of the actual trap instruction (`int 2E`).

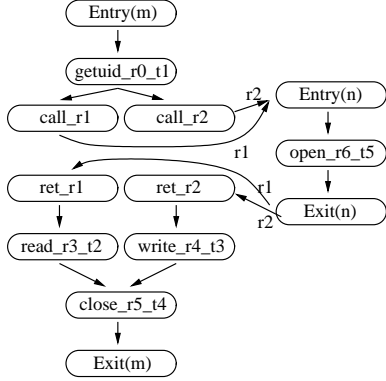


Figure 1: The CSFG model uses the return address chain to uniquely identify each system call site. For example, the system call site `C1` is identified by its return address `r1`. After the `getuid` call, the NFA moves the current state to `getuid_r0.t1`. When `open` is called, the NFA will move along the path beginning with `C1` only if the stack contains the address `r1`.

In CSFG, each function call is represented by a call node and a return node, such as `call_r1`, and `ret_r1`. Each call node or return node is labeled with its return address, such as `r1` and `r2` in Figure 1. The way that *BASS* uniquely identifies each system call site removes the non-determinism due to functions with multiple call sites. Despite the assignment of a unique coordinate to each system call site, CSFG is still an NFA, as illustrated by the functions `foo6` and `foo7` in Figure 2. Because of the `if` statement, `foo6` and `foo7` do not always make a system call. A function that may not always lead to any system call is referred to as a *may_function*. Because of *may_functions*, *BASS* cannot use a DFA traversal algorithm to traverse the CSFG.

Because the edges between per-function CSFGs are uniquely labeled by their return addresses, transitions between these CSFGs is always deterministic. Consequently, the CSFG traversal algorithm is a combination of DFA traversal, which is for inter-function traversal, and depth-first traversal, which is for intra-function traversal. Let's illustrate the basic concepts of this algorithm using the example in Figure 2. For a complete description of the CSFG traversal algorithm, please refer to [15]. Assume the current system call is `sys1`, which is legitimate, and the current CSFG cursor points to `sys1_r7.t1`. When a new system call `sys2` is called from the function `r9.t2`, if the

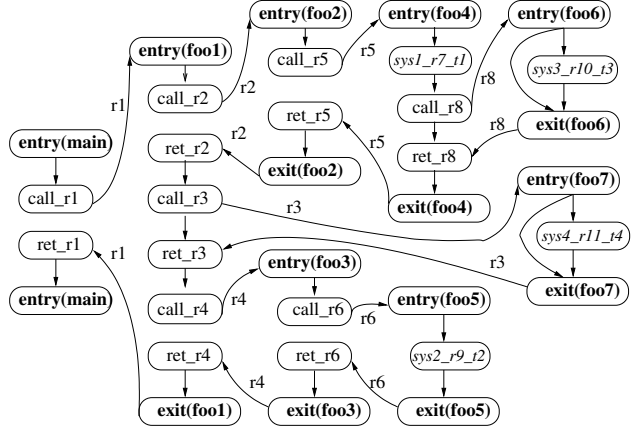


Figure 2: For the system call sequence $\{sys1, sys2\}$, when `sys2` is called, the `saved_stack` is $\{r1, r2, r5\}$, the `new_stack` is $\{r1, r4, r6\}$, and the `prefix` is $\{r1\}$. The run-time verifier needs to simulate the function returns and function calls to determine whether there is a path from the `saved_stack` to the `new_stack`.

CSFG traversal algorithm can successfully identify a path from the node `sys1_r7.t1` to the node `sys2_r9.t2` that does *not* contain any other system calls, `sys2` is considered legitimate and allowed to proceed.

When a new system call comes in, *BASS* first extracts the return address chain from the user stack. For example, when `sys2` is called, the return address chain is $\{r1, r4, r6, r9, t2\}$. The last two return addresses, `r9` and `t2`, are not used for graph traversal because they are used to identify the corresponding system call site. Therefore the CSFG traversal algorithm only uses $\{r1, r4, r6\}$, which is called `new_stack`. The `new_stack` of the last system call, `sys1` in this case, is called the `saved_stack`, and is $\{r1, r2, r5\}$.

The CSFG traversal algorithm first computes the `prefix` of the `saved_stack` and the `new_stack`, which is $\{r1\}$. Since the `saved_stack` is longer than the `prefix`, the application must have returned back to the function `foo1` before making the system call `sys2`. Each time the algorithm moves the cursor to a new function, it uses depth-first traversal to look for the exit node of the current function. This search is deterministic because every function has one and only exit node, and works correctly even when the CSFG contains *may_functions*, e.g., the `call_r8` node in `foo4`. The return address sequence after the `prefix` in the `save_stack` is $\{r2, r5\}$, based on which the algorithm performs the following operations to simulate function returns: (1) Find `exit(foo4)` using depth-first traversal; (2) Consume `r5` using DFA traversal, and move the cursor to `ret_r5`; (3) Find `exit(foo2)`

using depth-first traversal; (4) Consume `r2` using DFA traversal, and move the cursor to `ret_r2`.

After the above operations, the cursor is in the function `foo1`. Since the `new_stack` is longer than the prefix, the application must have made some function calls before invoking the system call `sys2`. Therefore, the algorithm needs to simulate the call operations. Each time the cursor moves to a new function, the algorithm uses depth-first traversal to look for the call node that is labeled with the current stack symbol. This operation is deterministic because each call node is uniquely labeled by its return address. The return addresses after the prefix in the `new_stack` is `{r4, r6}`, based on which the algorithm simulates the call operations using the following steps: (1) Find the call node labeled by `r4` using depth-first traversal, which is `call_r4`; (2) Consume `r4` using DFA traversal, and move the cursor to the callee of `call_r4`, which is `entry(foo3)`; (3) Find the `call_r6` node using depth-first traversal; (4) Consume `r6` using DFA traversal, and move the cursor to `entry(foo5)`. After completing the simulation of return and call operations, the CSFG algorithm uses depth-first traversal to reach the node `sys2_r9_r2`, which means the system call in question, `sys2`, is indeed legitimate.

Because of indirect calls (i.e. function pointers), even if an application’s source code is available, it is not always possible to construct a complete CSFG for that application. *BASS* solves this problem by inserting before every indirect call a `notify` system call, which informs the sandboxing engine the actual target of the indirect call. The sandboxing engine uses this information to temporarily connect two potentially disconnected CSFG components and continue CSFG traversal. The disadvantage of this approach is additional system call overhead for every indirect call.

3.2 Enhancements

In addition to checking the order of system calls and where they are invoked, *BASS* also checks the arguments of system calls to further reduce a program’s window of vulnerability. For each system call argument, *BASS* first computes a backward slice from it, and then performs *symbolic* constant propagation on the resulting slice to reduce it as much as possible. The reduction result could fall into one of the following three categories. First, the reduction result is a constant. This means that the value of the corresponding system call argument can be determined statically. In this case, this system call argument is a *static constant*. Second, the reduction result is not a constant but it depends only on input/configuration files, environment variables, or command line arguments, all of which are assumed to be immune from run-time tampering. The value of this type of

system call arguments can be determined after the initialization phase and never changes afterward. In this case, this system call argument is a *dynamic constant*. Third, the reduction result depends on inputs coming from the network at run time or real-time clocks. In this case, this system call argument is a *dynamic variable*.

For system call arguments that are static constants, the *BASS* compiler computes their values and include them in the system call model. For system call arguments that are dynamic constants, the *BASS* compiler determines the point in the program at which their value is fully determined, and inserts a `notify` call there to inform the run-time verifier of the value. For system call arguments that are static constant or dynamic constant, *BASS*’s run-time verifier should have their value before their corresponding system calls are invoked. For system call arguments that are dynamic variables, the *BASS* compiler tries to derive a partial constraint on them, e.g., a system call argument must be prefixed with a constant character string. Due to space constraint, the details of deriving system call argument constraints are left out. Interested readers can refer to the second author’s Ph.D. dissertation [15].

An application’s CSFG produced by *BASS* could help an attacker mounting a mimicry attack against the application. To mitigate this risk, *BASS* creates different CSFGs for different instances of the same application by randomly inserting `null` system calls into those functions that sit on the path between two consecutive system calls but they themselves do not lead to any system call. In addition, *BASS* inserts these system calls to an application at load time to eliminate the possibility that attackers correctly guess their existence. To prevent attackers from identifying these system calls through run-time disassembly, they are in the form of instructions with invalid op code or memory accesses that cause protection violation, rather than the the usual `int 2E` or `sysenter` instructions.

3.3 Graph Linking

Most previous work [28, 10, 16, 11] either required all libraries be statically linked or failed to handle dynamically linked libraries. Because Windows binaries use dynamically linked libraries (DLL) extensively, it is mandatory for *BASS* to sandbox DLLs as well. Because DLLs could be relocated when they are loaded, the return addresses or function addresses extracted statically must be adjusted accordingly at run time. Toward this end, *BASS* first calculates the base address of each DLL after it is loaded, and add the base address to the relative addresses statically extracted from the DLL.

To simplify the process of linking CSFGs, we applied the same idea used in dynamic linking by introducing a new

type of node called *trampoline* node. Statically, all calls to an import function are linked to its associated trampoline node, which also records the address of the import function's corresponding import address table entry. After all DLLs are loaded, the import address table entries are filled with the addresses of their associated functions. Therefore *BASS* can fill the trampoline nodes with their corresponding import address table entries. As in dynamic linking, fixing the trampoline nodes is all that is needed to link calling CSFGs with called CSFGs.

BASS handles Win32 executables and DLLs in nearly the same way. The only difference is that it takes special care in separating the read-write part of a DLL's CSFG to allow as much sharing of CSFGs as possible. On the Windows OS, by default the memory image of a DLL is shared by all processes that load it. If any process needs to modify a DLL, the OS will duplicate a copy of the modified pages for the modifying process according to the "copy on write" rule. When an application loads a DLL, it needs to modify the DLL's CSFG so that their return nodes point back to the application's call sites. Without special handling, this means many DLL CSFG pages need to be duplicated. To avoid this duplication, *BASS* rearranges the layout for each DLL's CSFG such that those nodes that need to be modified during graph linking, mainly the entry and exit nodes of exported functions, are stored in separate pages. Consequently, only these pages need to be duplicated, and the majority of DLL CSFGs still could be shared among applications.

3.4 Stack Walking

BASS uses the sequence of return addresses on the user stack as part of a system call site's coordinate. However, it is not trivial to identify where these return addresses are in general, because the use of frame pointer, typically EBP register, is not mandatory. Modern Windows compilers provide an optimization option that tries to use EBP as a regular register in order to improve program performance. For binaries produced by these compilers, it is no longer possible to pinpoint exactly the stack entries that contain return addresses. Our experiences show that many Win32 executables and DLLs indeed do away with the frame pointer, e.g., `KERNEL32.DLL`. This issue does not arise for *PAID* because *PAID*'s compiler is configured to use the frame pointer register when generating binary code. The Windows OS does provide a `stackwalk()` API to facilitate the debugging process. It could retrieve each frame on the stack by consulting the symbol information stored in PDB files. Unfortunately, most production-mode Win32 binaries do not come with a PDB file. Eventually, *BASS* chooses to maintain a shadow stack of return addresses. More specifically, it instruments each function call site to push the return

address before entering the callee, and pop the return address after returning from the callee. Consequently, *BASS*'s sandboxing engine can easily identify the return address sequence associated with an incoming system call. As a side effect, it also enhances the application security by detecting any stack smashing [19].

4 System Implementation

The system architecture of *BASS* is shown in Figure 3, and its various components are described in detail in the following subsections.

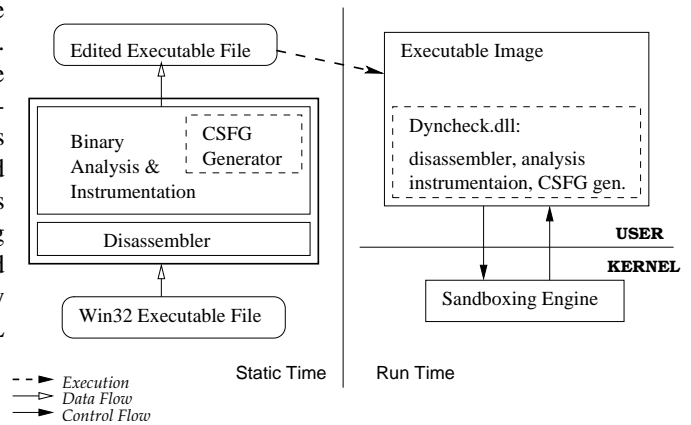


Figure 3: The system architecture of *BASS*, which consists of a static component that statically disassembles a binary file into instructions and extracts their system call model, a dynamic component that at run time disassembles those portions of the binary file that cannot be disassembled statically and extracts their system call model accordingly, and a sandboxing engine that compares an application's dynamic system call patterns with its system call model.

Most existing binary analysis and instrumentation tools are developed on Unix/Linux OS and/or RISC architecture, because it is generally easier to statically disassemble and analyze binaries on these platforms. However, Win32 binaries on the X86 architecture are much less susceptible to static disassembly and analysis, because of hand-crafted assembly routines and intentional obfuscation. To address this problem, we developed a new binary analysis/instrumentation system called *BIRD*, which performs both static and dynamic disassembly to guarantee that every instruction in a binary file will be properly examined before it is executed.

4.1 Binary Disassembly

In general, there are two main approaches to disassembling a binary file: *linear sweeping* and *recursive traversal*. Linear sweeping assumes every byte in the binary file is instruction and disassembles them one by one until it detects a disassembly error, e.g., when the leading byte of a supposed instruction does not correspond to any valid op code. Recursive traversal follows the control flow of an input binary starting from its main entry point, exploring both directions of each conditional branch instruction. Recursive traversal is in general more accurate than linear sweeping, but may suffer from the problem of low coverage due to indirect call or jump instructions.

Because the instructions that BIRD recovers from an executable binary are meant to be transformed, it is essential that BIRD's disassembler be 100% accurate. In contrast, commercial disassemblers such as IDA Pro are designed for reverse engineering purpose, and therefore do not have to be as accurate as BIRD. To overcome the fundamental limitations of static disassemblers with respect to Win32 binaries, BIRD adopts a hybrid architecture that statically disassembles a binary file as much as it can, and defers the rest to dynamic disassembly at run time. Because most of the instructions in a binary file are disassembled statically, the performance overhead of dynamic disassembling is minimal. However, the flexibility of dynamic disassembly offers a simple and effective fall-back mechanism for cases where static disassembling fails.

BIRD's static disassembler starts with a recursive traversal pass from the input binary's main entry point. Any instructions identified in this pass are guaranteed to be instructions. To improve the coverage of recursive traversal, BIRD applies data flow analysis to statically determine the target addresses of as many indirect jumps/calls as possible, and convert them into their direct counterparts. In addition, it exploits various PE header information such as export table, relocation table, etc., to identify places in a binary file that are known to be instructions.

The portions of a binary file that have been successfully disassembled are called *known regions*, whereas the rest are called *unknown regions*. Because of recursive traversal, the only way for a program's control to go from a known region to an unknown region is through an indirect control transfer instruction. Therefore, BIRD intercepts every indirect control transfer instruction at run time, and invokes the dynamic disassembler if it jumps to an unknown region. Run-time interception is through direct binary re-writing. This check-and-invoke logic forms the run-time engine of BIRD. The dynamic disassembler works similarly to the static one in that it also applies recursive traversal until the traversal encounters a known region or an indirect branch.

Kernel callbacks, including exception handlers, callbacks, and asynchronous procedure calls (APCs), are indirect calls coming from the kernel. Because they are invoked through a function pointer from a user-level library routine in `NTDLL.DLL` or `KERNEL32.DLL`, the fact that BIRD can intercept all indirect calls from these libraries means that it can intercept all kernel callbacks as well.

4.2 Binary Instrumentation

Because static disassembly cannot achieve 100% coverage, it is difficult to apply the traditional instrumentation strategy used in well-known binary instrumentation tools [17, 25], which start with extracting the input program's structure such as procedures and symbol table, and then merge the new code into it. To support binary instrumentation without complete knowledge of the program being instrumented, BIRD takes a local amendment approach, and performs both static and dynamic instrumentation. More concretely, BIRD adds a new section to the input program that contains the instrumentation code, and replaces the instruction at each instrumentation point with a jump to the corresponding instrumentation instruction sequence. There are two design issues in this approach. First, is it always possible to put a jump instruction at each instrumentation point? Second, how to ensure that the replaced instructions are executed in their original execution context?

In Intel X86 architecture, a jump instruction takes 5 bytes. If the instruction at the instrumentation point is shorter, e.g., a 2-byte short indirect branch, then it is necessary to replace multiple instructions. Instructions that are being replaced cannot be targets of direct branches. But it is OK if they are targets of indirect branches, because BIRD intercepts every indirect branch. If the length of the instruction at the instrumentation point is larger than or equal to 5 bytes, BIRD replaces the instruction directly; otherwise if none of the instructions following the instrumentation point are targets of direct branches, then BIRD replaces as many as possible to make room for the 5-byte jump; otherwise BIRD replaces the instruction at the instrumentation point with an `int 3` instruction, which is 2 bytes long. The `int 3` instruction generates a breakpoint exception, which is handled by BIRD's exception handler at run time. If the exception handler decides that a breakpoint exception occurs because of an `int 3` instruction BIRD inserted, it passes the control to BIRD's check-and-invoke logic, as if the control is passed from the instrumentation point directly. The `int 3` instruction is meant to be a fall-back mechanism when it is impossible to find enough bytes at the instrumentation point for the 5-byte jump instruction.

The values of registers and stack entries at the time when the control reaches an instrumentation point are saved away

before the check-and-invoke instrumentation code is called, and put back afterwards. Consequently, BIRD ensures that the replaced instructions are executed in the same context as if the instrumentation logic never takes place.

The check-and-invoke logic of BIRD is implemented as a DLL called `dyncheck.dll`, and is completely independent of the applications being instrumented. Moreover, once the import table of an instrumented program is modified, `dyncheck.dll` is automatically loaded at start-up time. Because the initialization routine of a DLL always gets control when the DLL is loaded, this enables BIRD to read in static information, such as known/unknown areas, and initialize required data structures before the program's main function starts. Because a program's import table may be immediately followed by some other data, it is not always possible to increase its size directly. To solve this problem, we keep the old import table, create a new import table that contains the original import table entries and any new entries we want to add, and modify the import table address field in the PE header to point to the new import table.

BASS leverages BIRD's binary instrumentation mechanism to build up the CSFGs for unknown areas as soon as they are converted into known areas. That is, once a section of instructions becomes known at run time, BIRD applies the same CSFG construction algorithm used statically to it, and constructs its corresponding CSFG.

4.3 Sandboxing Engine

Part of a program's CSFG is generated statically, and part of it is generated dynamically. When a program starts up, BASS reads in the static portion of its CSFG graph and fixes up its addresses. Then BASS links in each DLL's CSFG into the main CSFG. As new CSFGs are generated for statically unknown areas, they are also linked into the main CSFG. Although a program's CSFG may change at run time, BASS's sandboxing engine can still perform system call monitoring based on it, because BIRD guarantees that an instruction segment's CSFG must be available before it is executed.

4.3.1 Support for Multi-Threading

Supporting multi-threaded applications on the Win32 platform is relatively straightforward because each user thread corresponds to a kernel thread. On the Windows OS, there is a per-thread data structure called Thread Environment Block (TEB) to keep track of per-thread information such as thread id, thread stack base address and limit. This information is accessible to the corresponding kernel thread. To maintain a separate shadow stack for each thread, BASS

allocates a stack on the heap every time a new thread is created. This is possible because on Windows every time a thread is created, the initialization routine of every DLL, including `dyncheck.dll`, is invoked.

Because the code for pushing and popping return addresses to the shadow stack is supposed to be the same for all threads, it requires some address massaging to ensure that different threads are operating against different shadow stacks, even though they run the same code. BASS solves this problem by using thread local storage (TLS), which is a per-thread storage area. More specifically, a thread's TEB contains an array of pointers to its thread local storage regions. BASS reserves one element of this array to store the pointer to a thread's shadow stack across the entire application. Consequently, every thread can access its shadow stack using the same syntactic address, $TLS[X]$, where X is the index of the reserved element, even though they actually point to different stacks.

4.3.2 System Call Interception and Insertion

BASS intercepts system calls the same way as such tools as RegMon and FileMon [26], which are designed to monitor run-time behaviors of application programs. Modern Windows OSs include a kernel executive, which provides core system services. All user-level API calls such as those frequently used in `KERNEL32.DLL`, `NTDLL.DLL` will eventually call these system services or Native APIs. The kernel executive dispatches native API calls through the system service dispatcher table (SSDT). By writing a kernel device driver, BASS can modify the function pointer entries in SSDT and intercept all system calls with additional functions. Consequently, each time a system call is invoked, BASS's interception function is called first, which performs the required sandboxing operation and decides whether to block the system call.

5 Performance Evaluation

5.1 Methodology

The current BASS prototype can successfully run on Windows 2K, including Windows 2K Advanced Server, and Windows XP, with or without SP1 or SP2. Because BIRD needs to instrument known regions of executables and DLLs, we temporarily disable the Windows File Protection feature in order to modify the system DLLs and IIS. To evaluate the performance overhead of BASS, we measured the throughput and latency penalty of BASS with seven network server applications, which are briefly described in Table 5.2. Although BASS works on IE and Microsoft Office programs,

we don't use them in the performance study because it is difficult to accurately measure the performance overhead for interactive applications that require user actions. We ran each of these applications under the following four configurations: (1) Native mode, in which applications are executed without interception or checking, (2) BIRD Mode, in which applications are executed with BIRD's interception, (3) BIRD/BASS mode, in which applications are executed with BIRD's interception and BASS's system call checking, and (4) BIRD/BASS/Random mode, in which null system calls are randomly inserted into applications at load time and the resulting binaries are executed with BIRD's interception and BASS's system call checking. For this study, we chose 38 sensitive system calls to monitor that are related to file system and registry manipulation.

To test the performance of each server program, we used two client machines that continuously send 2000 requests to the test server applications. 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 creation of the first forked process and termination of the last forked process. The latency is calculated by taking the average of the response times for each of the 2000 requests. The server machine used in this experiment is a Windows XP SP1 machine with Pentium4 2.8GHz CPU and 256MB memory. One client machine is a 300-MHz Pentium2 with 128MB memory and the other client is a 1.1-GHz Pentium3 machine with 512MB memory. Both of them run Redhat Linux 7.2. The server and client machines are connected through a 100Mbps Ethernet link. 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 mail server, the clients retrieved a 1-KByte mail from 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.

5.2 Performance Overhead

Table 2 shows the throughput penalty of the test applications under the BIRD mode, BIRD/BASS mode, and BIRD/BASS/Random mode as compared with the Native mode. For most applications except GuildFTPd, the majority of the throughput penalty comes from BASS, which accounts for 1% to 7% drop in throughput, whereas BIRD accounts for between 0% to 3% throughput loss. The randomization component of BASS does not contribute much to throughput loss. With BIRD and BASS combined, the total throughput degradation is kept within 8%, a pretty

Application	Test Case	BIRD	Shadow Stack	CSFG Storage
Apache	fetch a 1KByte file	2.5%	178.7%	106.3%
BIND	query a name	2.5%	131.1%	270.0%
IIS W3 Service	fetch a file	3.47%	107.1%	238.1%
MTSEmail	send a 1KByte email	8.33%	108.34%	234.33%
Cerberus Ftpd	fetch a 1KByte file	4.17%	67.4%	161.0%
GuildFTPd	fetch a 1KByte file	4.24%	139.09%	120.5%
BFTelnetd	login and list files	6.25%	87.5%	207.8%

Table 1: The network server applications being used in the performance evaluation study, the test case for each of them, and the increase in their binary size under BASS.

Application	BIRD		BIRD+BASS		BIRD+BASS+Random	
Apache	99.9%	0.9%	94.2%	5.5%	94.0%	5.6%
BIND	97.8%	3.1%	92.3%	7.7%	91.9%	7.9%
IIS W3 Service	99.1%	1.1%	93.9%	6.3%	93.5%	6.8%
MTSEmail	99.7%	1.4%	97.3%	3.2%	97.3%	3.2%
Cerberus Ftpd	99.2%	1.2%	93.0%	7.6%	93.0%	8.2%
GuildFTPd	79.9%	25.3%	73.3%	32.7%	71.3%	33.2%
BFTelnetd	99.9%	1.5%	97.4%	3.4%	96.9%	3.5%

Table 2: The normalized throughput (left column) and latency penalty (right column) of the BIRD mode, the BIRD/BASS mode, and the BIRD/BASS/Random mode when compared with the Native mode for the seven test applications.

reasonable performance overhead. The overall throughput penalty of GuildFTPd is about 29%; 20% is due to BIRD and 9% due to BASS. The reason that GuildFTPd incurs a high BIRD-interception overhead is because it uses heavily dispatching functions and small callback functions, which correspond to indirect calls. As a result, the check-and-invoke logic in BIRD is triggered so frequently that eventually this logic accounts for a significant portion of GuildFTPd's overall run time.

The latency penalties for different applications running under different configurations are pretty similar to their throughput penalties. Overall, the latency penalty is also bounded under 8%, except GuildFTPd, whose latency penalty is more than 30%.

To give a detailed breakdown of BIRD's performance cost, Table 3 shows for each test application the coverage of BIRD's static disassembler, the static count of indirect control transfer instructions, the number of times the check-and-invoke logic is invoked at run time, and the number of times BIRD's dynamic disassembler is invoked. Because executables and DLLs are processed separately on their own, the reported numbers are for application binaries

Application	Static Coverage	No. of Static Indirect Branches	No. of Dynamic Indirect Branches	No. of Dynamic Disassembler Invocations
Apache	91%	109	3745	31
BIND	98%	390	18962	72
IIS W3 service	91%	125	12847	138
MTSEmail	99%	0	6352	0
Cerberus FTPd	79%	150	58764	263
GuildFTPd	83%	295	406196	89
BFTelnetd	80%	141	4459	136

Table 3: Detailed breakdown of BIRD’s static and dynamic disassembly overhead

themselves, excluding DLLs. The static coverage number is calculated by dividing the number of bytes that are known to be data or code statically over the entire binary size. Unsurprisingly, the check-and-invoke logic is triggered many more times in GuildFTPd than in other programs. This explains the high throughput penalty of GuildFTPd. However, the static disassembly coverage of Cerberus Ftpd is actually lower than GuildFTPd, even though its BIRD-related overhead is also much lower, under 1%. This demonstrates that the BIRD-related overhead is not necessarily determined by the number of times the dynamic disassembler is invoked. For example, Cerberus Ftpd invokes 263 times and GuildFTPd invokes only 89 times, and yet Cerberus Ftpd incurs lower overhead. Because GuildFTPd invokes the check-and-invoke logic so many times, its accumulative overhead becomes a significant overhead even though most of these checks confirm the target addresses point to a known area and therefore do not result in an invocation of the dynamic disassembler.

Table 1 shows the increase in binary size due to BIRD, shadow stack maintenance and storage of CSFG. BIRD’s contribution comes from the check-and-invoke logic and the dynamic disassembler, and is general quite small. The additional instrumentation required to maintain the shadow stack however increases the binary size significantly because it is designed to be thread-aware, and thus costs 62 bytes per function call in addition to some relocation logic. Finally, CSFG storage requires even more space than shadow stack maintenance, because the data structures are designed to provide sufficient flexibility to accommodate DLLs that are unknown statically. If one could assume that all DLLs are known in advance, it would be possible to develop a more compact representation for CSFGs and thus significantly reduce their storage space requirements.

To study the complexity of BASS’s graph traversal algorithm for real applications, we measured the number of nodes visited for each system call invocation when Apache and GuildFTPd are running under BASS. For Apache, the largest number of nodes that the graph traversal algorithm needs to visit when going from one system call to another

Application	Base Load Time (cycles)	BIRD Overhead	BIRD/BASS Overhead
Apache	84350072	23.08%	68.09%
BIND	112174792	54.84%	96.83%
IIS W3 service	215194865	45.16%	89.34%
MTSEmail	36329115	16.29%	42.98%
Cerberus Ftpd	47452796	10.76%	32.46%
GuildFTPd	150718358	30.51%	69.50%
BFTelnetd	123278084	10.67%	34.66%

Table 4: The increase in application start-up time introduced by BIRD and BIRD/BASS. The start-up delay is defined as the interval between when a binary is started and when its main entry point takes control.

is 54; in most cases the number of nodes visited is fewer than 10. The number of nodes visited per system call is more evenly distributed for GuildFTPd than for Apache: the largest number of nodes visited per system call is between 20 to 30, with most under 10. These results explain why the additional overhead introduced by BASS is relatively modest in practice, between 535 to 1600 CPU cycles, and demonstrate that the overhead of BASS’s graph traversal algorithm is indeed quite close to that of DFA traversal.

Table 4 shows the increase in application start-up time introduced by BIRD and BIRD/BASS, respectively. In general, BASS adds more start-up latency than BIRD because the former needs to read in the static portion of the application’s CSFG, and link the CSFGs of the DLLs with it to form the final CSFG, on which run-time system call monitoring is based. Although the increase in start-up latency is substantial, its practical impact is small as it is the sustained performance rather than the start-up time that matters for most network applications.

6 Attack Analysis and Limitations

When an attacker hijacks an application and steers the victim application’s control to a piece of injected code, BASS could immediately detect the attack because the injected code is in the data area and therefore not in the application’s unknown region. If the attacker steers the victim application’s control to an existing piece of code (e.g., a library function), BASS could detect the attack if the existing piece of code eventually makes a system call inconsistent with the application’s system call model.

The limitations of BASS stem from its system call graph model and binary interpretation mechanism. Like other compiler-based system call model extraction tools, BASS has zero false positive rate but could not completely eliminate all false negatives. A system call-based sandboxing system such as BASS cannot stop attacks that do not need to issue any additional system calls. For example, data at-

tacks [6] that modify a sensitive system call's arguments through buffer/integer overflowing can evade the detection of *BASS*. This problem can be somewhat alleviated through system call argument check, as is done in *PAID* [16]. When an attacker hijacks an application and the next legitimate system call is exactly what she needs to inflict damage, *BASS* is also completely powerless in this case. *BASS*'s ability to assign a unique coordinate to each system call site and check it at run time significantly reduces the possibility of mimicry attacks. This check makes it difficult to emulate legitimate system calls for a long period of time because it forces the application's control to go back to the application's code. More concretely, to attack *BASS*, the attacker needs to set up the user stack correctly according to the victim application's CSFG. Even if the attacker can do that, after making the first system call, the application's control will not return to the attacker's code since the control will go to whichever functions specified by the return addresses on the stack. However, more advanced mimicry attacks [14] try to apply data attacks to give the application's control back to the attacker during the emulation process, thus opening the possibility of defeating *BASS*'s coordinate check. Fortunately, *BASS*'s load-time randomization could potentially thwart this type of attacks as they require complete access to the application's binary.

Binaries do not come with type information, which in many cases can improve the accuracy of integrity checks. For example, from the source code of a network application, one can assume that all indirect function calls must go through either function pointers or special system routines such as signal handlers, and all function pointers should point to the entry points of some existing functions. From binaries, however, it is not always safe to equate an indirect call instruction to a function call using a function pointer. As a result, all the assumptions that come with function pointers may not hold for a given indirect call instruction.

Currently, *BIRD* cannot handle arbitrary self-modifying code or obfuscated code, although it can successfully execute self-decompressing programs that are compressed using tools such as *UPX* [27]. Although not an immediate concern, we expect more and more future applications may include self-modifying code either for performance optimization or for software protection. Therefore, we are currently investigating ways to enhance *BIRD* to support general self-modifying code.

7 Conclusion

To the best of our knowledge, *BASS* is the first system call-based sandboxing system that can automatically sandbox arbitrary Windows binaries running on the Intel X86 archi-

tecture without any human inputs and with low performance overhead, while achieving zero low false positive rate and very-close-to-zero false negative rate. Because *BASS* operates at the binary level, it is independent of the source languages and the associated compilers/linkers, and thus is applicable to a wide range of applications. In addition, *BASS* offers users an effective way to protect themselves from potential bugs in third-party applications without support from the original application developers or from special computer security vendors. More concretely, this work makes the following contributions:

- A highly accurate system call model representation that checks system call ordering, system call coordinates, and system call arguments that together greatly minimize the window of vulnerability to mimicry attacks.
- A flexible and efficient Win32/X86 binary interpretation system that has been shown to correctly interpret a wide variety of Windows applications, including Microsoft Office suite and IIS, that state-of-the-art disassemblers fail to disassemble completely.
- One of the most if not the most comprehensive system call pattern-based host-based intrusion detection systems that could automatically and accurately sandbox applications that involve dynamically linked libraries, multi-threading, and exception handlers.

References

- [1] M. Abadi, M. Budiu, I. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353, Alexandria, VA, November 2005.
- [2] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of 1997 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1997.
- [3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
- [4] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.
- [5] B. D. Bus, D. Kastner, D. Chanet, L. V. Put, and B. D. Sutter. Post-pass compaction techniques. *Commun. ACM*, 46(8):41–46, 2003.
- [6] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of 14th USENIX Security Symposium*, August 2005.

- [7] A. Conry-Murray. Product focus: Behavior-blocking stops unknown malicious code. <http://www.networkmagazine.com/shared/article/showArticle.jhtml?articleId=8703363&classroom=> (2002).
- [8] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, page 194, Berkeley, CA, May 2004.
- [9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.
- [10] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79. USENIX Association, 2002.
- [11] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of The 11th Annual Network and Distributed System Security Symposium*, Feb. 2004.
- [12] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the USENIX Security Symposium*, July 1996.
- [13] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, 2002.
- [14] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, August 2005.
- [15] L. C. Lam. Program transformation techniques for host-based intrusion prevention. *Ph.D. dissertation, Computer Science Department, Stony Brook University*, December, 2005.
- [16] L. C. Lam and T. cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Seventh International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, September 2004.
- [17] J. R. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, CA, June 1995.
- [18] S. Nanda, W. Li, L. chung Lam, and T. cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the 4th IEEE/ACM Conference on Code Generation and Optimization (CGO'06)*, March 2006.
- [19] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceeding of the 2003 Usenix Annual Technical Conference*, June 2003.
- [20] V. Prevelakis and D. Spinellis. Sandboxing applications. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 119 – 126, 2001.
- [21] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, Tsukuba, Japan, Nov. 2005.
- [22] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using etch. In *The USENIX Windows NT Workshop Proceedings*, Seattle, Washington, August 1997.
- [23] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.
- [24] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Notice*, 39(4):528–539, 2004.
- [25] A. Srivastava and D. W. Wall. A practical system for inter-module code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.
- [26] SysInternals. <http://www.sysinternals.com/ntw2k/source/regmon.shtml>.
- [27] UPX. The ultimate packer for executables. <http://upx.sourceforge.net>.
- [28] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, 2001.
- [29] 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*, pages 255–264, Washington, DC, USA, 2002. ACM Press.