Reining In Windows API Abuses with In-lined Reference Monitors

Kevin W. Hamlen Vishwath Mohan Richard Wartell

University of Texas at Dallas
Technical Report UTDCS-18-10
{hamlen,vishwath.mohan,richard.wartell}@utdallas.edu

Abstract

Malware attacks typically effect damage by abusing operating system resources (e.g., the file system) that are exposed via system API calls. A method of using automated binary code-rewriting to monitor API calls and their arguments is presented and evaluated. Unlike traditional monitoring approaches, the framework requires no modification of the operating system, has no effect upon trusted processes, and preserves the behavior of most complex x86 native code binaries generated by mainstream compilers, including binaries that are object-oriented, graphical, contain callbacks, and use a mixture of static and dynamic linking. A separate verifier certifies that rewritten binaries cannot circumvent the monitor at runtime, allowing the binary-rewriter to remain untrusted. An implementation for Microsoft Windows demonstrates that the technique is effective and practical for real-world systems and architectures.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Access controls; D.3.4 [*Programming Languages*]: Processors—Code generation; D.2.4 [*Software Engineering*]: Software/Program Verification—Validation

General Terms Languages, Security

Keywords binary rewriting, sandboxing, software fault isolation

1. Introduction

Reference monitoring is a well-established paradigm for enforcing safety policies by mediating access to security-relevant system resources. For example, most modern operating systems implement reference monitors that limit access to file system resources based on access control lists. Unfortunately, reference monitors implemented at the operating system level have the disadvantage that they cannot be easily modified by third parties to enforce new policies. Doing so typically requires access to operating system source code that may be proprietary and therefore unavailable, or write access to restricted operating system binaries. Even when these are available, modified operating systems can be difficult to maintain across OS revisions and patches, and the modifications can potentially impact even trusted processes, causing compatibility problems.

An attractive alternative is *In-lined Reference Monitoring* [19], wherein a *binary rewriter* in-lines the reference monitor directly into the binary code of untrusted processes. In-lined Reference Monitors (IRM's) can be implemented without modifying the operating system, and with minimal system privileges (e.g., write access to the untrusted program file but not to the system binaries). Furthermore, IRM's constrain the behavior of untrusted processes without affecting the behavior or performance of other processes, and are therefore well-suited for enforcing specialized, application-specific policies.

However, most binary rewriting algorithms must make strong assumptions about untrusted code in order to successfully preserve the behavior of policy-adherent programs and prevent policy-violating behavior of malicious programs. For example, most IRM systems rely upon control-flow and data encapsulation properties imposed by an underlying virtual machine to prevent malicious code from circumventing in-lined security guards or modifying injected IRM variables. For this reason, a majority of IRM implementations target type-safe bytecode languages, such as Java (e.g., [2, 3, 9]), ActionScript [22], or .NET CIL [11], which impose such constraints at the VM level. This makes these systems inapplicable to the detection and prevention of most conventional malware, since the vast majority of malware is expressed as untyped native code. Applying IRM technology to untyped domains has therefore remained a longstanding challenge in the field.

We address this challenge through the design and implementation of an IRM framework that targets native x86 machine code binaries on Microsoft Windows systems. The framework automatically transforms native code binaries so as to redirect system API calls through a trusted *policy-enforcement library*. The policy-enforcement library therefore has access to all API calls and their arguments before (and after) they are serviced, and can use this information to enforce arbitrary safety policies over histories of these security-relevant events. The binary-rewriting algorithm is carefully crafted so as to accommodate a large class of difficult features commonly found in real x86 executables, including computed jumps, dynamic linking, code sections that interleave executable code with static data, and untrusted callback functions invoked by the operating system. Experiments show that it successfully preserves the behavior of non-malicious, real-world Windows binaries obtained from a variety of different mainstream compilers.

A small, trusted verifier shifts the significant complexity of the rewriting system out of the trusted computing base by independently certifying that rewritten binaries cannot circumvent the policy-enforcement library. Thus, binaries that pass verification are guaranteed to be safe to execute. While reflective code can change its behavior in response to rewriting, the verifier ensures that any such behavioral changes cannot effect policy-violations.

Section 2 begins with an overview of the system architecture, including a discussion of various features of x86 Windows binaries that are important for understanding the system implementation. The binary-rewriting and verification algorithms that are the focus of the research are presented in Section 3. Experimental results that demonstrate the practicality of the approach are presented in Section 4, followed by a more systematic treatment of the system assumptions and limitations in Section 5. Finally, Section 6 discusses related work and Section 7 concludes.

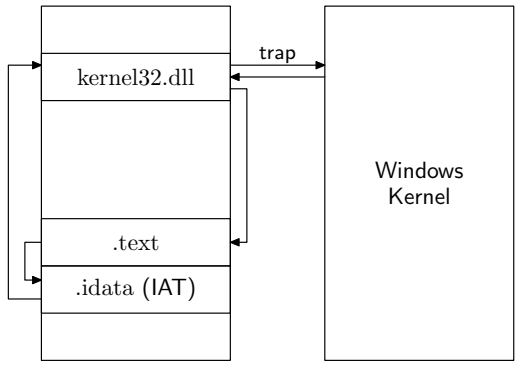

Figure 1. Windows API call procedure

2. System Overview

Attackers in our model submit arbitrary x86 binary code for execution on victim systems. Neither attackers nor defenders are assumed to have kernel-level (ring 0) privileges on the system. Attacker-supplied code runs with user-level privileges, and must therefore leverage kernel-supplied services to perform malicious actions, such as corrupting the file system or accessing the network to divulge confidential data. The defender's ability to thwart these attacks stems from his ability to modify attacker-supplied code before it is executed. His goal is therefore to reliably monitor and restrict access to security-relevant kernel services without the aid of kernel modifications or malware source code, and without impairing the functionality of non-malicious code.

Windows binaries typically access protected operating system resources like the file system and network using a procedure depicted in Figure 1. First, user code calls a public accessor function exported by a system library such as `kernel32.dll`. The user code learns the entrypoint address of the accessor function by consulting an *import address table* (IAT) constructed by the system loader at process start. Both the IAT and the system library reside in the address space of the untrusted process. The system library runs at the same privilege level as the untrusted process into which it is loaded, so it cannot service the request directly. Instead, it implements a system-specific protocol that culminates in a trap to the operating system kernel. The kernel grants (or rejects) the request using its elevated privileges, and then transfers control back to the caller.

One popular technique for monitoring system calls of user processes is *IAT hooking* (c.f., [12, 18]), wherein IAT entries are replaced with the addresses of guard functions that mediate access to the system-supplied accessor functions. Unfortunately, malware can easily circumvent this form of monitoring. For example, malware can call the system's accessor functions directly without using the IAT, either by guessing their entrypoints or obtaining them from a variety of public sources (e.g., by implementing the same algorithm used by the system loader to build the IAT). Alternatively, malware can simply implement the kernel trap directly, avoiding the accessor functions entirely.

Statically identifying these unsafe operations is provably undecidable in general. The core difficulty is the ubiquity of computed jump instructions, which permeate almost all native code binaries. Deciding whether a computed jump instruction will unsafely bypass the IAT at runtime requires statically inferring the program register and memory state at arbitrary code points, which is well known to be equivalent to the halting problem. Moreover, since x86 instructions are unaligned, computed jumps make it impossible to reliably identify all instructions in an untrusted binary program; disassemblers must heuristically guess the addresses of many instruction sequences in order to generate a complete disassembly. Malware binaries are often specifically crafted to defeat these heuristics, thereby concealing malicious instruction sequences from analysis tools.

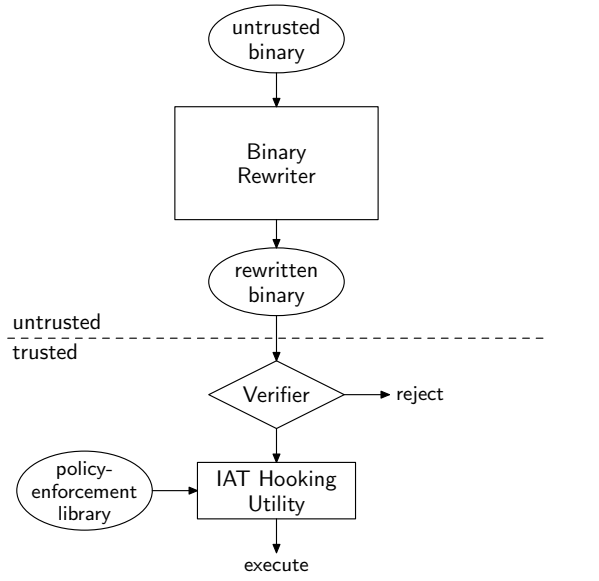

Figure 2. System architecture

Our *Rewriting and In-lining System* (REINS) applies a binary-rewriting algorithm that automatically transforms untrusted x86 programs so as to force all access of kernel services to use the process's IAT. An IAT hooking utility can then be used to reliably guard all accesses of security-relevant kernel services. The behavior of code that unsafely bypasses the IAT or that contains hidden instruction sequences is not necessarily preserved by the transformation process, but the behavior of code that adheres to a certain minimal set of standard x86 conventions obeyed by mainstream compilers (detailed in Section 5) is preserved. The transformation process produces a new binary in which all instruction sequences are provably memory-aligned. This allows a separate verifier to easily disassemble rewritten binaries and independently prove that they contain no reachable trap instructions or jumps that bypass the IAT hooks.

The rewriting algorithm uses *software fault isolation* (SFI) [24] to constrain control flows of untrusted code. *Data Execution Prevention* (DEP) hardware [13, 25] is leveraged to enforce memory safety. Self-modifying code and executable data, which are both rejected by DEP-enabled operating systems, are not supported.

The resulting system architecture is illustrated in Figure 2. Untrusted binaries are first analyzed and transformed into safe binaries by a binary rewriter. Our binary rewriter is implemented as an IDAPython [4] program that leverages the considerable analysis power of the Hex-rays IDA Pro commercial disassembler to automatically identify function entrypoints and distinguish code from data in complex x86 binaries. While IDA Pro is powerful, it is not perfect; malicious code can defeat the analysis to conceal instruction sequences as data or disguise function entrypoints. A separate verifier therefore certifies that rewritten binaries are policy-adherent. Malicious binaries that defeat the rewriter's analysis might result in rewritten binaries that fail verification or that fail to execute properly, but never in policy violations.

3. Binary Rewriting Algorithm

3.1 Control-flow Safety

Our binary rewriting algorithm is based on a particularly elegant SFI approach implemented in the PittSFIeld system [16], which partitions x86 instruction sequences into c-byte *chunks*. Chunk-spanning instructions and targets of jumps are moved to chunk boundaries by padding the instruction stream with `nop` (no-operation) instructions. This serves three purposes:

- When c is a power of 2, computed jumps can be easily confined to chunk boundaries by guarding them with an instruction that dynamically clears the low-order bits of the jump target.
- Confining all jumps to chunk boundaries makes chunks atomic. Thus, co-locating guard instructions and the instructions they guard within the same chunk prevents circumvention of the guard by a computed jump. A chunk size of $c = 16$ suffices to contain each guarded instruction sequence in our system.
- Aligning chunks to c-byte boundaries allows a simple, linear disassembler to reliably discover all reachable instructions in rewritten programs and verify that all computed jumps are suitably guarded.

While it is not possible in general to statically identify all jump targets in arbitrary binary code, modern commercial disassemblers employ a variety of powerful heuristics that suffice to identify jump targets (though not necessarily the specific jumps that target them) in most non-malicious code. A failure to identify one or more jump targets may lead to rewritten code that does not execute properly, but the verifier ensures that it cannot lead to a policy violation.

To allow trusted, unrewritten system libraries to safely coexist in the same address space as chunk-aligned, rewritten binaries, we logically divide the virtual address space of each untrusted process into *low memory* and *high memory*. Low memory addresses range from 0 to $d - 1$ and may contain rewritten code sections and non-executable data sections. High memory addresses range from d and up, and may contain code sections of trusted libraries and arbitrary data sections.

Partition point d is chosen to be a power of 2 so that a single guard instruction suffices to elegantly confine untrusted computed jumps to chunk boundaries in low memory. For example, a jump that targets the address currently stored in the `eax` register can be guarded as follows:

```
and eax, (d − c)
jmp eax
```

This clears both the low-order and high-order bits of the target address before jumping, preventing untrusted code from jumping directly to a system accessor function or to a non-chunk boundary in its own code. The logical partitioning of virtual addresses into low and high memory is feasible in practice because rewritten code sections are generated by the rewriter and can therefore be positioned in low memory, while trusted libraries are relocatable through *rebasing* and can therefore be moved to high memory when necessary.

The above scheme suffices to constrain the control-flow of untrusted executables, but it will cause most programs with computed jumps to malfunction. Computed jump destinations are obtained from a variety of obscure sources in real x86 programs, including static data stored within the code section, dynamic method dispatch tables created at runtime, and even external data files. Identifying and rewriting all such pointers is not usually feasible even for the most powerful disassembly tools. The guard instructions described above will therefore typically result in redirection of some of these pointers to arbitrary chunk boundaries, causing erratic behavior and usually a crash (but not a policy violation). To our knowledge, all existing SFI systems based on binary-rewriting require external information to accommodate such computed jumps—usually information obtained from application source code.

To overcome this limitation, our system takes the novel approach of converting the original untrusted code section into a lookup table that maps old code addresses to corresponding new code addresses. That is, the entrypoint of each original function in the old code section is overwritten with a pointer to the entrypoint of the corresponding rewritten function in the new code section. To allow rewritten code to efficiently distinguish these pointers from real code, each is tagged with a leading byte that cannot appear as the first byte of valid code. We use a tag byte of `0xF4`, which encodes an x86 `hlt` instruction that is illegal in protected mode.

This yields the concise guard instruction sequence given in the first row of Table 1. Here, a conditional move instruction (`cmovz`) dynamically detects and repoints old code pointers to new, rewritten code at runtime. While the entire instruction sequence fits in one 16-byte chunk, only the final two instructions must share a chunk to enforce the control-flow safety policy.

Retaining the old code section as a data section has the additional advantage of retaining any static data that may be interspersed amongst the code. This data can therefore be read by the rewritten executable at its original addresses, avoiding many difficult data preservation problems that hamper other SFI systems. The tradeoff is an increased size of rewritten programs, which tend to be around twice the size of the original. However, this does not necessarily lead to an equivalent increase in runtime process sizes. Our experiences with real x86 executables indicates that dynamic data sizes tend to eclipse static code sizes in memory-intensive processes. Thus, in most cases rewritten process sizes incur only a fraction of the size increase experienced by the disk images whence they were loaded.

When the original jump instruction employs a memory operand, the rewritten code requires one scratch register, as shown in row 2 of Table 1. Scratch registers are in very short supply on x86 architectures; however, the `eax` register is caller-save by convention and is not used to pass arguments by any calling convention supported by any mainstream x86 compiler [8]. This makes it well-suited as a scratch register at these callsites. Code whose behavior depends on the value of `eax` at callsites might malfunction after applying the code transformation in row 2, but will not result in a policy violation. No other transformations in Table 1 require scratch registers.

A particularly common form of computed jump deserves special note. Return instructions (`ret`) jump to the address currently stored on the top of the stack (and optionally pop n additional bytes from the stack afterwards). These are guarded by the instruction given in row 3 of Table 1, which masks the return address atop the stack to a low memory chunk boundary. Call instructions are moved to the ends of chunks so that the return addresses they push onto the stack are always aligned to the start of the following chunk. Thus, the return guards have no effect upon return addresses pushed by properly rewritten call instructions, but protect against buffer overrun attacks that might overwrite the return address with an unsafe destination.

To allow untrusted code to safely access trusted library functions in high memory, the rewriter permits one form of computed jump to remain unguarded. Computed jumps whose operands directly reference the IAT are retained as-is. Such jumps usually have the following form:

```
call [.idata:n]
```

where `.idata` is the section of the executable reserved for the IAT and n is an offset that defines which IAT entry is being used. These jumps are safe since an IAT hooking utility can ensure that they always target policy-compliant addresses at runtime.

Description	Original code	Rewritten code

Computed jumps (registers)	`call/jmp` r	`cmp byte ptr [`r`], 0xF4`
		`cmovz` r`, [`r`+1]`
		`and` r`,` $(d-c)$
		`call/jmp` r

Computed jumps (memory)	`call/jmp [`m`]`	`mov eax, [`m`]`
		`cmp byte ptr [eax], 0xF4`
		`cmovz eax, [eax+1]`
		`and eax,` $(d-c)$
		`call/jmp eax`

Returns	`ret` (n)	`and [esp],` $(d-c)$ `ret` (n)

IAT loads	`mov` rm`, [.idata:`n`]`	`mov` rm`, offset trampoline_`n
		
		`trampoline_`n`:`
		`and [esp],` $(d-c)$
		`jmp [.idata:`n`]`

Tail-calls to high memory	`jmp [.idata:`n`]`	`and [esp],` $(d-c)$ `jmp [.idata:`n`]`

Callback registrations	`call/jmp [.idata:`n`]`	`call/jmp trampoline_`n
		
		`trampoline_`n`:`
		`push` ⟨*callback registration function address*⟩
		`call intermediary.reg_callback`
		
		`return_trampoline:`
		`call intermediary.callback_ret`

Dynamic linking	`call [.idata:GetProcAddress]`	`push offset trampoline_pool`
		`call [.idata:GetProcAddress]`
		
		`trampoline_pool:`
		`.ALIGN` c
		`call intermediary.dll_out`
		`.ALIGN` c
		`call intermediary.dll_out`
		⋮

Table 1. Summary of x86 code transformations

Unfortunately, not all uses of the IAT have this simple form. Most x86-targeting compilers also generate code that loads addresses from the IAT into registers and later performs computed jumps through those registers rather than reading the IAT directly. This alternative form is employed to more efficiently encode multiple calls to the same import using a shorter instruction sequence.

To safely accommodate such calls, the rewriter identifies and modifies all instructions that use IAT entries as data. An example of such an instruction is given in row 4 of Table 1. For each such instruction, the rewriter replaces the IAT memory operand with the address of a unique *trampoline chunk* introduced to the rewritten code section. The trampoline chunk performs a safe jump to the trusted function using a direct IAT reference. Thus, any use of the replacement argument value as a jump target results in a jump to the trampoline chunk, which has the same effect as invoking the desired function directly.

3.2 Memory Safety

To prevent untrusted binaries from dynamically modifying code sections or executing data sections as code, untrusted processes are executed with DEP enabled. DEP-supporting operating systems and hardware allow memory pages to be marked non-executable (NX). Attempts to execute code in NX pages result in runtime access violations. The binary rewriter sets the NX bit on the pages of all low memory sections other than rewritten code sections to prevent them from being executed as code.

User processes on Windows systems can change the NX bit on memory pages within their own address spaces, but this can only be accomplished via a small collection of system API functions—most notably `VirtualProtect` and `VirtualAlloc`. IAT hooking is applied to replace the IAT entries of these functions with trusted wrapper functions. The wrapper functions silently set the NX bit on all pages in low memory other than rewritten code pages. The wrappers do not require any elevated privileges; they simply access the real `VirtualProtect` and `VirtualAlloc` system functions with modified arguments. These system functions are accessible to trusted libraries but not untrusted libraries because the trusted libraries have separate IAT's that do not undergo IAT hooking.

Trusted libraries can therefore use the `VirtualProtect` and `VirtualAlloc` API functions to protect their local heap and stack pages from untrusted code that executes in the same address space. They do so by enabling read and/or write access to relevant pages on entry to the trusted code and revoking said access on exit. IAT hooks prevent rewritten code from directly accessing these same functions to reverse these effects. This prevents the rewritten code from gaining unauthorized access to trusted memory.

Our memory safety enforcement strategy conservatively rejects untrusted, self-modifying code. Such code is increasingly rare outside of malware, both because it is incompatible with DEP-enabled systems and because it typically incurs a high performance penalty. However, it still appears regularly in certain specialized application domains such as JIT-compilers and certain self-extracting installers. Future work concerns dynamically invoking the rewriter and/or verifier to support such runtime code generation.

3.3 Verification

The verifier certifies that rewritten programs cannot circumvent the IAT and are therefore policy-adherent, but it does not prove that the rewriting process is behavior-preserving. This reduced obligation greatly simplifies the verifier relative to the rewriter, resulting in a much smaller trusted computing base.

To verify control-flow safety, the rewriter disassembles each executable section in the untrusted binary and verifies the following properties:

- All executable sections reside in low memory.
- All exported symbols (including the program entrypoint) target low memory chunk boundaries. (This is necessary to prevent an untrusted module from importing an unsafe address into its IAT from another untrusted module at load time.)
- No instruction spans a chunk boundary.
- All static branches target low memory chunk boundaries.
- All computed `call`, `jmp`, and `ret` instructions that do not use the IAT as a memory operand are immediately preceeded by a suitable masking instruction that sanitizes the target address. The masking instruction resides in the same chunk as the jump it guards.
- Computed jump instructions that read the IAT access a properly aligned IAT entry, and are preceded by a suitable mask of the return address. (Call instructions must *end* on a chunk boundary rather than requiring a mask instruction, since they push their own return addresses.)
- Kernel trapping instructions, such as `int` and `syscall` instructions, do not appear in the disassembly.

These properties ensure that hidden, unaligned instruction sequences that might be concealed within untrusted code sections are not reachable at runtime. This allows the verifier to limit its attention to a linear disassembly of executable sections, avoiding any reliance upon the incomplete code-discovery heuristics needed by more sophisticated disassemblers to produce full disassemblies for arbitrary binaries.

3.4 Callbacks

Callbacks are a ubiquitous component of almost all real-world x86 binaries. These occur when an untrusted binary passes a code pointer to a trusted library, and this code pointer is later used by the trusted library as a jump destination. Since trusted libraries cannot always be rewritten to guard these computed jumps (e.g., because the library is not write-accessible or may contain kernel trap instructions unsupported by the rewriter), this creates an obvious loophole through which untrusted code could violate control-flow safety. Effectively enforcing control-flow safety in practical settings therefore requires a mechanism for guarding callbacks.

The most common form of callback is a simple return address, which is used by every trusted callee to return to its caller. To prevent untrusted callers from providing unsafe return addresses to trusted callees, the verifier requires a chunk-aligned, low memory address atop the stack during any control transfer from low to high memory. This is achieved by the `and` instructions in rows 3 and 5 of Table 1, which mask the return address before jumps to trusted callees.

Other callback pointers are passed to trusted callees as explicit function arguments. These turn out to be more common than one might expect in real x86 binaries. For instance, even toy C programs compiled for Windows internally register a callback function that is invoked by the system at process termination to set the process's return code. Adequately supporting such callbacks educes two challenges: (1) The protection system must ensure that the callback addresses passed to trusted *callback registration functions* are legal jump targets. (2) There must exist a mechanism by which callbacks can safely return to trusted callers—a jump from low to high memory that would normally be disallowed by the protection mechanism.

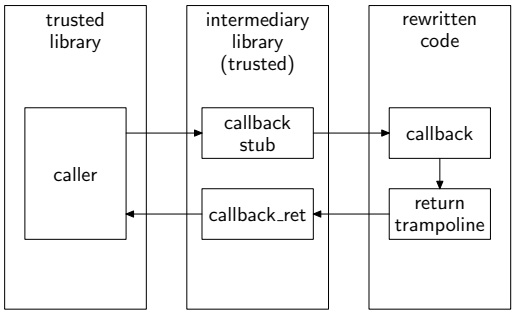

Figure 3. Control-flow for untrusted callbacks and returns

In our framework, these challenges are surmounted using a trusted *intermediary library* that facilitates these control-flows. The rewriter redirects calls of trusted callback registration functions to the trampolines shown in row 6 of Table 1, which invoke the intermediary library. The intermediary library modifies relevant code pointer arguments on the stack[1] to point to unique *callback stubs* within its own address space, and saves the original (untrusted) callback addresses in a local data structure before passing control to the real registration function. This has the effect of registering the stubs as the callbacks instead of the original callbacks.

When the system attempts to invoke the callback, the modified control-flow depicted in Figure 3 results. The trusted library calls the stub that was registered by the intermediary library, which passes control through to the original callback it replaced. However, before doing so the stub replaces the trusted caller's return address on the stack with the address of a *return trampoline* within the rewritten code section. This prevents the untrusted callee from attempting to return directly to a high memory address (which would be prevented by the masking instruction that guards untrusted returns). When the callback completes, it returns to the return trampoline, which transfers control to the intermediary's `callback_ret` function. This function is the dual of the callback stub; it retrieves the stored return address of the trusted caller and safely returns control there.

In all of the above, the intermediary library must avoid trusting any input provided by the rewritten code. All code pointers are therefore masked to low memory chunk boundaries before use, and any trusted code pointers provided by the trusted caller must be stored in protected memory pages to prevent corruption by rewritten code.

3.5 Dynamic Linking

Dynamically linked programs load libraries and compute their function entrypoints at runtime. These dynamically linked functions are not traditionally accessed through the IAT. Instead, system-supplied API functions are employed to load new libraries into the local address space and discover the addresses of their exported functions at runtime. On Windows the API functions are named `LoadLibrary` and `GetProcAddress`, respectively.

Safely preserving the behavior of dynamically linked programs poses three interesting challenges for our SFI system: (1) The system must arrange for the return value of `GetProcAddress` to

[1] Such arguments are located at known stack offsets or within known fields of structures passed to the registration function by reference.

be a chunk boundary in low memory (since otherwise control-flow guards will prevent its use as a jump target). (2) The code at this returned address must pass control through to the dynamically linked function without knowing the function's identity at code-generation time. (3) The system must prevent rewritten code from using dynamic linking to gain direct access to prohibited library functions, such as `VirtualProtect`, that should be redirected to safe wrapper functions.

To accomplish this, the IAT hooking mechanism overwrites the IAT entry for `GetProcAddress` with the entrypoint of a trusted replacement. The replacement `GetProcAddress` reserves a unique member of a *trampoline pool* located in the rewritten code section and returns its address instead of the address of the requested function. Calls to dynamically linked functions therefore get routed through the trampoline pool instead of directly to the dynamically linked function.

The trampoline pool is shown in the final row of Table 1, and consists of a series of chunk-aligned calls to another trusted intermediary function named `dll_out`. These calls do not return; the implementation of `dll_out` pops the return address as an argument to determine which trampoline was called. It then passes control through to the dynamically linked function for which it reserved that trampoline, rerouting any calls of prohibited functions to safe wrapper functions when necessary.

As with callbacks, the intermediary library must be carefully designed to distrust any input supplied by the rewritten code. The implementation therefore ensures that any call to `dll_out` results in control flowing to the entrypoint of a non-prohibited, exported library function. In addition, the rewriter-supplied trampoline pool must be large enough for the intermediary library to allocate one unique trampoline for each unique function that undergoes dynamic linking at runtime. While the number of dynamically linked functions cannot be reliably determined statically, in practice it is almost always equal to the number of distinct `GetProcAddress` call instructions in the original program. A deficient guess causes the replacement `GetProcAddress` to return a null pointer once the trampoline pool has been exhausted, which usually results in an error message and premature termination of the process.

4. Experiments

To evaluate the binary rewriting algorithm presented in Section 3 we developed an implementation of the system for the 32-bit version of Microsoft Windows XP/Vista. The implementation consists of four components: a rewriter, a verifier, an IAT hooking utility, and an intermediary library (see Sections 3.4 and 3.5). None of the components require elevated privileges to perform their functions. While the implementation is Windows-specific, we believe that the general approach could be implemented for any modern operating system that supports DEP technology.

The rewriter transforms Windows Portable Executable (PE) files in accordance with the algorithm presented in Section 3. Its implementation consists of about 2000 lines of IDAPython code that executes atop the Hex-rays IDA Pro 5.2 commercial disassembler. One of IDA Pro's primary uses is as a malware reverse engineering and de-obfuscating tool, so it boasts many powerful code analyses that heuristically recover program structural information without assistance from a code-producer. These analyses are leveraged by our system to automatically distinguish code from data and identify function entrypoints to facilitate the rewriting algorithm.

In contrast to the significant complexity of the rewriting infrastructure, the verifier's implementation consists of 1500 lines of OCaml code that uses no external libraries or utilities (other than the built-in OCaml standard libraries). Of these 1500 lines, approximately 1000 are devoted to x86 instruction decoding, 300 to PE binary parsing, and 200 to the actual verification algorithm described in Section 3.3. The decoder handles the entire x86 instruction set, including floating point, MMX and all SSE extensions documented in the Intel and AMD manuals. We found this to be necessary for practical testing since production-level binaries frequently contain at least some exotic instructions. (A good example is the frequent use of quad-word MMX instructions to more efficiently zero-fill data structures.) No code is shared between the verifier and rewriter.

The intermediary library consists of approximately 450 lines of C and hand-written, in-lined assembly code that facilitates callbacks and dynamic linking. An additional 150-line configuration file itemizes all trusted callback registration functions exported by Windows libraries used by the test programs, along with the stack locations of any code pointer arguments they take as input. We supported all callback registration functions exported by `comdlg32`, `gdi32`, `kernel32`, `msvcrt`, and `user32`. Information about exports from these libraries was obtained by examining the C header files for each library and identifying function pointer types that appear in exported function prototypes and structs.

The IAT hooking utility is charged with replacing the IAT entries of all prohibited system functions imported by rewritten PE files (e.g., `VirtualProtect`) with the addresses of trusted replacement functions. In addition, it must add the intermediary library to the PE's list of imported modules. This can be problematic if it becomes necessary to expand the size of the PE header, since doing so would shift the positions of the binary sections that follow it, resulting in misplaced data and incorrect code.

We discovered an elegant solution to this problem in the form of a mere 4-byte change to PE headers that accomplishes all of the above. Our IAT hooking utility simply changes the library name `kernel32.dll` in the headers of rewritten binaries to `helper32.dll` (the name of our intermediary library). This causes the system loader to draw all IAT entries previously imported from `kernel32.dll` from the export table of the intermediary library instead. The intermediary library is compiled so as to export all symbols found in `kernel32.dll` as forwards[2] to the real `kernel32.dll`. Prohibited functions, such as `VirtualProtect`, are not forwarded; they are exported as local replacement functions of the same name. Our intermediary library thus doubles as the policy-enforcement library. (To separate them, one could simply code the intermediary library so as to statically load a separate policy-enforcement library.)

We tested our system on the set of binary programs listed in Table 2. All experiments were performed on a 1.8G dual-processor AMD Opteron 248 with 2G of ram running Windows XP Professional. The second column lists file sizes, which doubled on average. The third column indicates the effect of rewriting on the size of the code segment; the transformations detailed in Section 3 increased code segments by a bit less than half on average. Columns four and five list the time necessary for rewriting and verification. We made no particular attempt to optimize the rewriter implementation so rewriting times are somewhat high, but verification times are more realistic. Verification speed was about 0.9s per megabyte of code on average. The final two columns report median runtimes (for non-interactive programs) and process sizes for some medium-to-large sample inputs. Runtimes typically increased by less than 5%, and process size typically increased by less than about 40%.

Several of the test programs are CPU benchmarking utilities (e.g., `whetstone`). The runtimes of most of these actually improved marginally after rewriting. After some analysis, we concluded that this is because the rewriting algorithm word-aligns all jump targets and prevents any instruction's encoding from crossing a word boundary. This improves the effectiveness of look-ahead

[2] Forwards are not wrapper functions. A forward link in a PE export table causes the loader to link directly to the module and address to which the forward points, incurring no runtime overhead.

Program	File Sizes (K)	Code Sizes (K)	Rewriting Time	Verification Time	Runtimes (s)	Process Sizes (K)
notepad	68 / 108 (+59%)	30 / 41 (+37%)	6.7s	45ms	N/A (interactive)	3736 / 3700 (-1%)
iexplore*	91 / 98 (+8%)	7 / 7 (+0%)	1.7s	14ms	N/A (interactive)	22584 / 22400 (-1%)
jar	69 / 138 (+100%)	52 / 70 (+35%)	15.4s	63ms	4.15 / 4.38 (+5%)	1840 / 2590 (+41%)
gcc	88 / 174 (+98%)	62 / 88 (+42%)	30.2s	86ms	5.51 / 5.48 (+0%)	1732 / 2336 (+35%)
g++	90 / 179 (+99%)	64 / 91 (+42%)	31.7s	157ms	22.00 / 22.50 (+2%)	2281 / 2236 (-2%)
objcopy	700 / 1390 (+98%)	503 / 741 (+47%)	25min	664ms	0.55 / 0.58 (+5%)	1576 / 2444 (+55%)
size	496 / 994 (+100%)	347 / 510 (+47%)	10min	452ms	0.11 / 0.12 (+8%)	1712 / 2560 (+50%)
strings	495 / 994 (+101%)	347 / 510 (+47%)	10min	482ms	10.58 / 11.02 (+4%)	2140 / 3000 (+40%)
windres	602 / 1170 (+94%)	420 / 615 (+46%)	16min	542ms	0.25 / 0.27 (+8%)	2408 / 3288 (+36%)
dllwrap	62 / 101 (+63%)	28 / 40 (+43%)	7.5s	43ms	0.33 / 0.33 (+0%)	1644 / 2208 (+34%)
c++filt	544 / 1070 (+97%)	384 / 568 (+48%)	14min	503ms	0.06 / 0.07 (+11%)	1336 / 2104 (+57%)
ar	513 / 1000 (+95%)	335 / 523 (+56%)	11min	461ms	0.01 / 0.01 (+0%)	1412 / 2404 (+70%)
dlltool	596 / 1160 (+95%)	416 / 612 (+47%)	16min	542ms	N/A (negligible)	840 / 1240 (+48%)
whetstone	27 / 38 (+41%)	8 / 11 (+44%)	0.9s	15ms	816.50 / 806.90 (-1%)	1364 / 2096 (+54%)
linpack	32 / 50 (+58%)	14 / 19 (+36%)	30min	20ms	249.80 / 248.30 (-1%)	1676 / 2416 (+44%)
pi_ccs5	130 / 293 (+125%)	129 / 167 (+29%)	74.0s	142ms	58.94 / 60.05 (+3%)	38428 / 39180 (+2%)
median	(+96%)	(+42%)	74.0s	150ms	(+2%)	(+40%)

*Only the exe part of this application was rewritten, not its helper libraries.

Table 2. Experimental results

and instruction decoding piplining optimizations implemented by modern processors.

The experimental results reported in Table 2 enforced only the core access control policies required to prevent control-flow and memory safety violations. For example, unsafe callback registrations and callbacks were intercepted and rerouted via the intermediary library to prevent control-flow violations. However, we also used the system to enforce simple audit policies that logged API calls and their arguments to a disk file (which was useful for debugging), and access control policies that denied access to various API functions based on caller-supplied arguments. These marginally increased runtimes (mostly due to disk accesses for the log file operations) but did not noticeably affect any of the other statistics.

The set of programs we were able to effectively rewrite was mainly constrained by the need to find programs that do not use the Windows COM automation library, which our system does not yet support. COM is unfortunately used to some degree by almost every production-level Windows binary, so this was a significant restriction. We were able to rewrite COM-aware programs, but using their COM features resulted in a crash. For example, we could rewrite the exe portion of Microsoft Internet Exporer 6 but not its helper libraries, since the helper libraries use COM. COM support is discussed in greater detail in Section 5.

5. Discussion

The binary rewriting algorithm presented in Section 3 only preserves the behavior of executables that adhere to certain x86 code generation conventions. Code that violates these conventions might yield rewritten code that fails verification or fails to execute properly, but will never lead to verified code that violates control-flow safety. Nevertheless, the practicality of the approach depends on its ability to preserve the behavior of a large class of non-malicious code. Compatibility limitations of this sort have been a major obstacle to widespread adoption of much past SFI research.

In this section we discuss the binary code conventions that are prerequisites for behavior-preservation under our algorithm and argue that these conventions are satisfied by code generated by most mainstream x86 compilers for major source languages. Known limitations of our algorithm are highlighted during the course of the discussion.

5.1 Code Pointers

The binary rewriting algorithm in Section 3 expects each code pointer used as a jump target by untrusted code to originate from one of five possible sources:

- a low-memory address drawn from the instruction pointer (e.g., a return address pushed by a call instruction),
- static data that points to a function entrypoint in the original binary,
- a function entrypoint stored in the IAT,
- a return address pushed by a trusted caller during a callback, or
- a return value yielded by the system's dynamic linking API (e.g., GetProcAddress).

The semantics of computed jumps whose arguments do not originate from one of these sources are not necessarily preserved by the rewriting algorithm. For example, a program that computes a jump destination using pointer arithmetic will usually end up jumping to an arbitrary chunk boundary, resulting in erratic behavior.

Low-level source languages (e.g., C/C++) that expose code pointers to the programmer do not typically make any guarantees about the relative values of such pointers. For example, Microsoft Visual C compilers return function pointers that reference jump tables instead of the actual function bodies, and the ordering of the jump table contents varies across builds. Thus, aside from reflective code that is designed to fail when modified (usually for anti-piracy purposes), dependence upon raw pointer values is rare even in large-scale, production-level binaries. We have encountered only one significant exception to this in our analysis of Windows binaries, detailed below.

Microsoft *Component Object Model* (COM) [17] is a binary-level standard by which Windows programs interoperate by exchanging *interfaces* expressed as arrays of function pointers. Since these function pointers do not originate from one of the sources listed above, programs that use COM are not semantically preserved by our current implementation; rewritten programs that attempt to use COM services usually crash.

Supporting COM requires the development of a COM wrapper library that automatically converts interfaces to arrays of chunk-aligned trampoline pointers, similar to our mechanism for supporting dynamic linking (see Section 3.5). The IAT hooking mechanism

can then redirect COM calls to the wrapper library to support the safe exchange of COM interfaces. The development effort required to produce a fully functional COM wrapper library is potentially significant due to the plethora of COM services that must be supported, but not theoretically challenging. It is reserved for future work.

Almost all large-scale commercial Windows programs make moderate to extensive use of COM services at runtime. Thus, lack of COM support is presently the most significant compatibility limitation of our implementation. The experiments in Section 4 therefore limit their attention to programs that do not use COM, or whose use of COM is restricted to particular application functionalities that can be avoided during testing.

5.2 Distinguishing Code from Data

Binaries generated by most mainstream compilers mix code and static data within the `.text` section of the executable. The static data usually consists of pointer tables or string literals. The rewriter presented in Section 3 relies upon a classification algorithm that heuristically distinguishes the code from the data. If code is misclassified as data, that code is incorrectly omitted from the rewritten binary's code section and erratic behavior results. If data is misclassified as code that contains function entrypoints, the rewriter might overwrite some of the data with function pointers as it constructs the lookup table described in Section 3.1. This can result in corruption of the static data.

As part of the classification process, the algorithm must also identify all possible targets of computed jumps. The vast majority of these targets are function entrypoints, which are readily identifiable in most binaries by the characteristic function prologues and epilogues that begin and conclude most function bodies. The few remaining computed jump targets are gleaned through a combination of code reachability analysis and pattern-matching heuristics that identify instruction sequences compiled from common source language structures (e.g., switch-case statements) that often compile to computed jumps.

In practice we find that IDA Pro's automatic binary analysis works quite well for most of the above, requiring only occasional manual adjustments. The most common manual adjustment is to identify a code block that the analysis misclassified as data. All manual adjustments performed during our experiments were fairly simple and could have been automated with IDA Pro plug-ins.

A more subtle prerequisite of the rewriting algorithm requires that all computed jump destinations in the original binary be at least $w + 1$ bytes away from the next computed jump destination or any following data, where w is the word size of the system. This is necessary to ensure that there is sufficient space for the rewriter to write a tagged code pointer at that address without overwriting any adjacent code pointers or data. In practice this is a reasonable requirement, since most computed jump destinations are already 16-byte aligned for performance reasons, and since all binaries compatible with *hotpatching* technology (c.f., [21]) have at least $w + 1$ bytes of padding between consecutive function entrypoints. As a result of these conventions, we have not encountered any violations of this prerequisite during testing.

5.3 Register Usage Conventions

The 32-bit x86 architecture has a very limited register store which is aggressively allocated by most compilers during code generation. As a result, it is not possible to reliably reserve any dedicated registers for SFI use throughout the lifetime of arbitrary x86 code. Most binary rewriting algorithms proposed by past SFI research require at least one dedicated register, which introduces significant compatibility problems when scaling to large-scale, real-world x86 binaries.

The rewriting algorithm presented in Section 3 requires no dedicated registers and only one temporary scratch register for use at call sites with memory operands, and at return sites that cross from high to low memory. At call sites the `eax` register is the only general-purpose register that is both caller-save and not used for argument-passing by any standard x86 calling convention [8]. Since it is caller-save by convention, code whose behavior depends on preserving it across calls typically saves and restores it before and after each call. This makes it well-suited as a scratch register. At return sites, `eax` is not a good choice because it is used by most calling conventions to store return values. Instead, our implementation uses `edx`, which is the only other available caller-save register.[3]

The x86 execution environment also includes a collection of status flags that are written by comparison instructions and read by conditional branches. The rewriting algorithm assumes that program behavior does not depend on preserving these flags across computed jumps. Status flags are extremely volatile, so to our knowledge no compiler generates code whose behavior depends on preserving them across computed jumps; however, in theory this limitation could be lifted by saving and restoring the flags before and after in-lined guard instructions.

5.4 Multi-threading

To safely support multi-threaded programs, rewritten programs require access to a threading implementation that provides certain minimal memory-isolation between threads that share a virtual address space. Without such isolation, chunks are not atomic—a context switch within a chunk allows the new thread to modify the register and stack contents of the suspended thread, potentially effecting a control-flow violation when that thread resumes.

The default Windows threading implementation does not provide the isolation necessary to protect multi-threaded programs from such attacks. It stores almost all thread state, including saved stack and register contents, within the local address space of the process, where it can potentially be located and corrupted by a malicious thread. This leaves our current implementation vulnerable to concurrency attacks.

Closing this vulnerability requires development of a safe, trusted threading library that sets and clears appropriate memory access permissions for locally stored thread state during context switches. IAT hooks could then be used to reroute standard Windows threads calls to the corresponding safe replacements. The Windows API exposes all the functionality necessary to implement such a library via its *fibers* interface, so this solution would not require elevated system privileges to implement. Building such a library is a project that we intend to pursue in future work.

5.5 Trusted Libraries

As trusted components of the system, trusted libraries are another source of potential vulnerabilities. Our system does not defend against attacks that exploit existing vulnerabilities in trusted code, such as deficient checks of user inputs that may lead to buffer overflows within trusted modules, or unsafe storage of security-relevant data in places where it could be corrupted by an attacker.

Instead, the system is intended to allow as many modules as possible to be shifted to the untrusted half of Figure 2. That is, any library that does not need to implement direct kernel traps can potentially be rewritten and executed as untrusted code, so that its accesses of kernel services can be monitored by the policy-enforcement library. Any exploitation of such rewritten libraries by an attacker cannot bypass the monitoring mechanism, and therefore

[3] The `ecx` register is also caller-save by convention, but some object-oriented compilers reserve it for holding the *this* pointer.

cannot do significant damage to the system. Ideally, any remaining trusted libraries consist almost entirely of small, stateless trap instruction sequences that simply dispatch service requests to the kernel. Such simple libraries are unlikely to include many exploitable vulnerabilities, and therefore constitute a more acceptable trusted computing base.

6. Related Work

SFI research has traditionally concerned the problem of safely and efficiently executing a small, untrusted program (e.g., a browser plug-in or operating system extension) within the address space of a larger, trusted program. The SFI system enforces control-flow and memory safety by inserting extra instructions into untrusted code to guard potentially dangerous jumps and memory accesses. The seminal work on the subject by Wahbe, Lucco, Anderson, and Graham [24] implements such module encapsulation for RISC architectures. Their original technique relies upon several features specific to RISC, including memory-aligned instruction encodings and a large register store that affords the reservation of dedicated registers.

Implementing effective SFI for CISC architectures is generally viewed to be a substantially more difficult problem. Most CISC SFI implementations simplify the many difficult design hurdles by targeting only x86 assembly language programs yielded by a specific compiler. Using assembly code instead of raw binaries, and limiting attention to the output of a particular compiler (and often only output yielded by particular compiler options), allows a simple static analysis to infer a table of all legal jump targets. (The jump targets are usually explicitly labeled in the assembly code output produced by the compiler.) This table can then be injected into the rewritten code as a runtime data structure that is dynamically consulted to guard computed jumps. Examples of this approach include the MiSFiT [20] and SASI [5] systems, both of which target gcc assembly code output.

The success of these systems has led to a large body of practical IRM research (e.g., [1–3, 7, 11, 14, 22]) for type-safe virtual machine languages such as Java bytecode. These systems leverage the control-flow and memory safety properties enforced by the Java virtual machine and bytecode verifier to inject non-circumventable guard instructions into untrusted bytecode binaries. While this has proved very effective for enforcing complex, application- and system-specific security policies over well-typed binary code, it does not address the problem of protecting systems against malware expressed as untyped native code. Since the vast majority of current-day malware threats propagate as untyped x86 native code binaries, this is a significant open problem.

More recently, two different works have offered more generalized CISC SFI solutions that rely on binary meta-data rather than compiler-generated assembly code listings. Microsoft's XFI system [6] enforces control-flow safety by inserting a unique byte sequence (encoded as the unused operand of a multi-byte `nop` instruction) immediately before all valid jump destinations in rewritten code. This allows guard instructions to dynamically test for the existence of this reserved sequence before permitting a computed jump to the following address. The PittSFIeld system [16], which inspired our work, introduces the chunk-partitioning strategy described at the beginning of Section 3.1. Both systems include formal proofs of safety; the safety proof for XFI is written out for human consumption whereas the one for PittSFIeld is machine-checkable and uses a verifier similar to the one described in Section 3.3.

The XFI and PittSFIeld systems are significant steps forward because they offer more elegant CISC SFI strategies than past work and because they include formal verification. However, both still require the cooperation of code-producers because they rely on information not typically found in real COTS binaries. For example, XFI is implemented atop the Vulcan library [23], which identifies jump destinations and code pointers by drawing upon a *program database* (PDB) file produced by many compilers for debugging purposes, but that is not usually released to the general public. No prior work to our knowledge has successfully applied SFI technology to x86 COTS applications using only information available to the average code-consumer. In addition, while the PittSFIeld verifier works effectively for most gcc-generated machine code, it does not support certain problematic control flows introduced by IAT calls, operating system callbacks, and dynamic linking, which appear regularly in COTS binaries for x86 Windows systems.

The increasing availability of hardware DEP support [13, 25] over the past few years has introduced a significant shift in the tradeoffs that have motivated most past SFI research. DEP provides a hardware-supported (and therefore relatively light-weight) memory and control-flow safety enforcement mechanism that operates at page rather than process granularity. It therefore offers an extremely elegant SFI enforcement paradigm, but only if untrusted code can be prevented from accessing and abusing the DEP services themselves to change memory permissions and thereby violate safety. This observation has led us to a reversal of the traditional SFI paradigm: Where past SFI work typically prevents a small, untrusted module from damaging a larger, trusted program that contains it, our work seeks to protect smaller, trusted modules (system libraries) from larger, untrusted processes that import them.

7. Conclusion

We have presented the design and implementation of an IRM Rewriting and In-lining System (REINS) that monitors and restricts Windows API calls of untrusted native x86 binary executable programs. The framework employs an array of software fault isolation techniques to constrain control-flows of untrusted code, and leverages Data Execution Prevention technology available in modern hardware and operating systems to enforce memory safety. These are applied to force untrusted code to follow the standard Windows API calling protocol when accessing system resources, allowing such accesses to be reliably monitored by a trusted policy-enforcement library.

The software fault isolation mechanism consists of a binary rewriting algorithm that requires no explicit cooperation from code-producers, and that is behavior-preserving for a large class of COTS binaries drawn from several mainstream compiler families. To our knowledge, no past SFI work has achieved this. Since the rewriting infrastructure is of significant complexity, a smaller, separate verifier automatically certifies that rewritten code produced by the rewriter cannot circumvent the monitor. This allows the rewriting task to be shifted to an untrusted third party if desired.

Experiments on COTS applications showed that although rewriting doubled file sizes on average, the median process size increased by less than half, and runtime overhead was less than 5% for typical programs. The most significant compatibility problem encountered during testing involved lack of support for the Windows COM automation library, which is used by most large-scale Windows programs for at least a few program features. Supporting COM is therefore an important subject of future work.

The current Windows implementation of the system remains vulnerable to certain multi-threading attacks in which a malicious thread corrupts the stored state of a suspended thread before the suspended thread resumes. This vulnerability could be closed by linking rewritten programs to a safer threads implementation that isolates threads that share the same address space from each other's stored stack and register state. The existing SFI framework provides

the foundation necessary to implement such isolation, but this is a project we have not yet undertaken. Development of a safe Windows threads library for this purpose is another important practical avenue of future work.

The experiments reported here focus on testing the soundness, transparency, and feasibility of the binary rewriting algorithm on a real-world operating system. Less attention was devoted to enforcing complex security policies; enforced policies were limited to simple audit and access control. Past work [10, 15] has shown that IRM systems are capable of enforcing a much richer class of policies that includes history-based, temporal properties. Developing policy-enforcement libraries that implement such policies is therefore a logical next step toward applying our framework to interesting, practical security problems for these real-world systems.

References

[1] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. In *Proc. Workshop on Run Time Enforcement for Mobile and Distributed Systems*, pages 45–58, 2007.

[2] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. ACM Programming Language Design and Implementation*, pages 305–314, 2005.

[3] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550, 2005.

[4] G. Erdélyi. IDAPython: User scripting for a complex application. Bachelor's thesis, EVTEK University of Applied Sciences, 2008.

[5] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, 1999.

[6] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. Symposium on Operating Systems Design and Implementation*, pages 75–88, 2006.

[7] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proc. IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[8] A. Fog. *Calling Conventions for different C++ compilers and operating systems*. Copenhagen University College of Engineering, 2009.

[9] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *Proc. ACM Workshop on Programming Languages and Analysis for Security*, 2008.

[10] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Programming Languages and Systems*, 28(1):175–205, 2006.

[11] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. ACM Workshop on Programming Languages and Analysis for Security*, pages 7–16, 2006.

[12] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*, chapter Chapter 4: The Age-Old Art of Hooking, pages 73–74. Pearson Education, Inc., 2006.

[13] Intel Corporation. Execute disable bit and enterprise security. `www.intel.com/technology/xdbit`.

[14] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[15] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of non-safety policies. *ACM Trans. Information and System Security*, 12(3), 2009.

[16] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. 15th USENIX Security Symposium*, August 2006.

[17] Microsoft Corporation. COM: Component object model technologies. `www.microsoft.com/com`.

[18] J. Raber and B. Krumheuer. QuietRIATT: Rebuilding the import address table using hooked DLL calls. In *Black Hat Technical Security Conference*, Washington, D.C., January 2009.

[19] F. B. Schneider. Enforceable security policies. *ACM Trans. on Information and Systems Security*, 3(1):30–50, 2000.

[20] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *Proc. USENIX Annual Technical Conference*, pages 41–54, 1996.

[21] A. Sotirov. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conference*, Las Vegas, Nevada, August 2006.

[22] M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *Proc. Verification, Model Checking, and Abstract Interpretation*, 2010. to appear.

[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] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *Operating System Review*, 27(5):203–216, 1993.

[25] A. Zeichick. *Security Ahoy! Flying the NX Flag on Windows and AMD64 To Stop Attacks*. Advanced Micro Devices, March 2007.