Securing Untrusted Code via Compiler-Agnostic Binary Rewriting*

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin
Department of Computer Science, The University of Texas at Dallas
800 W. Campbell Rd, Richardson, TX, 75080
{richard.wartell, vishwath.mohan, hamlen, zhiqiang.lin}@utdallas.edu

ABSTRACT

Binary code from untrusted sources remains one of the primary vehicles for malicious software attacks. This paper presents REINS, a new, more general, and lighter-weight binary rewriting and inlining system to tame and secure untrusted binary programs. Unlike traditional monitors, REINS requires no cooperation from code-producers in the form of source code or debugging symbols, requires no client-side support infrastructure (e.g., a virtual machine or hypervisor), and preserves the behavior of even complex, event-driven, x86 native COTS binaries generated by aggressively optimizing compilers. This makes it exceptionally easy to deploy. The safety of programs rewritten by REINS is independently machine-verifiable, allowing rewriting to be deployed as an untrusted third-party service. An implementation of REINS for Microsoft Windows demonstrates that it is effective and practical for a real-world OS and architecture, introducing only about 2.4% runtime overhead to rewritten binaries.

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*

1. INTRODUCTION

Software is often released in binary form. There are numerous distribution channels, such as downloading from the vendor's web site, sharing through a P2P network, or sending via email attachments. All of these channels can introduce and distribute malicious code. Thus, it is very common for end-users to possess known but not fully trusted binary code, or even unknown binaries that they are lured to run. To date, there are two major classes of practical mechanisms to protect users while running such binaries. One is a heavy-weight approach that runs the binary in a contained virtual machine (VM) (e.g., [13, 32, 25]). The other is a lighter-weight approach that runs them in a sandboxing environment with an in-lined reference monitor (IRM) [36, 30, 39].

*This work was supported in part by NSF award #1054629 and AFOSR awards FA9550-08-1-0044 and FA9550-10-1-0088.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

The VM approach is appealing for several reasons. First, it avoids the problem of statically disassembling CISC binaries. Instead, VMs dynamically translate binary code with the aid of just-in-time binary translation [13, 32, 25, 18]. This allows dynamically computed jump targets to be identified and disassembled on the fly. Second, VMs can intercept API calls and filter them based on a security policy. Third, even if damage occurs, it can potentially be contained within the VM. Therefore, VM approach has been widely used in securing software and analyzing malicious code.

However, production-level VMs can be extremely large relative to the untrusted processes they guard, introducing significant computational overhead when they are applied to enforce fine-grained policies. Their high complexity also makes them difficult to formally verify; a single bug in the VM implementation leaves users vulnerable to attack. Meanwhile, there is an air-gap if the binary needs to access host files, and VM services must also bridge the semantic-gap [7]. While lighter-weight VM alternatives, such as program shepherding [18], lessen some of these drawbacks, they still remain larger and slower than IRMs.

On the other hand, a large body of past research (e.g., SFI [36], PittSFIeld [20], CFI [1], XFI [10], NaCl [39]) has recognized the many advantages of client-side, static, binary-rewriting for securing untrusted, mobile, native code applications. Binary-rewriting boasts great deployment flexibility since it can be implemented separately from the code-producer (e.g., by the code-consumer or a third party), and the rewritten code can be safely and transparently executed on machines with no specialized security hardware, software, or VMs. Moreover, it offers superior performance to many VM technologies since it statically in-lines a light-weight VM logic directly into untrusted code, avoiding overheads associated with context-switching and dynamic code generation. Finally, safety of rewritten binaries can be machine-verified automatically (in the fashion of proof-carrying-code [22]), allowing rewriting to be performed by an untrusted third party.

Unfortunately, all past approaches to rewriting native binary code require some form of cooperation from code-producers. For example, Google's Native Client (NaCl) [39] requires a special compiler to modify the client programs at the source level and use NaCl's trusted libraries. Likewise, Microsoft's CFI [1] and XFI [10] requires code-producers to supply a *program database* (PDB) file (essentially a debug symbol table) with their released binaries. Earlier works such as PittSFIeld [20] and SASI [11] require code-producers to provide `gcc`-produced assembly code. Code that does not satisfy these requirements cannot be rewritten and is therefore conservatively rejected by these systems. These restrictions have prevented binary-rewriting from being applied to the vast majority of native binaries because most code-producers do not provide such support and are unlikely to do so in the near future.

Therefore, in this paper we present the first, purely static, CISC

native code rewriting and in-lining system (REINS) that requires no cooperation from code-producers (i.e., is compiler-agnostic). Unlike past work, REINS can automatically rewrite large-scale, COTS, Windows applications yielded by arbitrary compilers, even with no access to source-level information (e.g., PDB files or debug symbol stores). It transparently supports a large category of production-level applications unsupportable by past efforts, including those that include event-driven OS-callbacks, dynamic linking, exceptions, multithreading, computed jumps, and mixtures of trusted and untrusted modules. In past efforts we have successfully used REINS' core rewriting engine to implement basic block randomization for over 100 Windows and Linux applications without source code [37].

Realizing REINS for COTS x86 binary in Windows platform raises many challenges, including semantic preservation of dynamically computed jumps, code interleaved with data, function callbacks, and imperfect disassembly. We address these challenges through the design and implementation of a suite of novel techniques, including conservative disassembly and indirect jump target identification. Central to our approach is a binary transformation strategy that expects and tolerates many forms of disassembly errors by conservatively treating every byte in target code sections as both code and static data. This obviates the need for perfect disassemblies, which are seldom realizable in practice without source code.

To tame and secure unsafe logic inside the binary code, REINS automatically transforms binaries to redirect system API calls through a trusted *policy-enforcement library*. The library thereby mediates all security-relevant API calls and their arguments before (and after) they are serviced, and uses this information to enforce safety policies over histories of these security-relevant events. Indirect control flow transfers (e.g., `call/jmp/ret`) are protected by in-lined guard code that ensures that they target safe code addresses when executed. In addition, a small, trusted verifier shifts the significant complexity of the rewriting system out of the trusted computing base (TCB) by independently certifying that rewritten binaries cannot circumvent the in-lined monitor. Thus, binaries that pass verification are guaranteed to be safe to execute. While reflective code can change its behavior in response to rewriting, verification ensures that such changes cannot effect policy-violations.

In summary, REINS makes the following contributions:

- We present the first *compiler-agnostic, machine-certifying, x86 rewriting* algorithm that supports real-world COTS binaries without any appeal to source code or debug symbols. To the best of our knowledge, all past static binary rewriting techniques require source-level or debugging information to support many COTS binary features.
- We design a set of novel techniques to support binary families for which fully correct automated disassembly is provably undecidable, including those that contain computed jumps, dynamic linking, static data interleaved with code, and untrusted callback functions invoked by the OS.
- We have implemented REINS as a proof-of-concept prototype, and tested it on a number of binaries including malware code. Our empirical evaluation shows that our system successfully preserves the behavior of non-malicious, real-world Windows applications, introducing runtime overheads of about 2.4%.

2. BACKGROUND AND OVERVIEW

2.1 Background

Assumptions. The goal of our system is to tame and secure malicious code in untrusted binaries through static binary rewriting. Since a majority of malware threats currently target Windows x86 platforms, we assume the binary code is running in Microsoft Windows OS with x86 architecture. Protecting Linux binary code is outside the scope of this paper. (In fact, rewriting Windows binary code is much more challenging than for Linux due to the much greater diversity of Windows-targeting compilers.)

Our goal is to design a compiler-agnostic static binary rewriting technique, so we do not impose any constraints on the code-producer; it could be any Windows platform compiler, or even hand-written machine code. Debug information (e.g., PDB) is assumed to be unavailable. Like all past native code IRM systems, our fully static approach rejects attempts at self-modification; untrusted code may only implement runtime-generated code through standard system API calls, such as dynamic link library (DLL) loading. Code-injection attacks are therefore thwarted because the monitor ensures that any injected code is unreachable.

In addition, our goal is not to protect untrusted code from harming itself. Rather, we prevent modules that may have been compromised (e.g., by a buffer overflow) from abusing the system API to damage the file system or network, and from corrupting trusted modules (e.g., system libraries) that may share the untrusted module's address space. This confines any damage to the untrusted module.

Threat model. 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. 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 application source code, and without impairing the functionality of non-malicious code.

Attacks. The central challenge for any protection mechanism that constrains untrusted native code is the problem of taming computed jumps, which dynamically compute control-flow destinations at runtime and execute them. Attackers who manage to corrupt these computations or the data underlying them can hijack the control-flow, potentially executing arbitrary code.

While computed jumps may seem rare to those accustomed to source-level programming, they actually pervade almost all binary programs compiled from all source languages. Computed jumps typically include returns (whose destinations are drawn from stack data), method calls (which use method dispatch tables), library calls (which use import address tables), multi-way branches (e.g., switch-case), and optimizations that cache code addresses to registers.

Deciding whether any of these jumps might target an unsafe location at runtime requires statically inferring the program register and memory state at arbitrary code points, which is a well known undecidable problem. Moreover, since x86 instructions are unaligned (i.e., any byte can be the start of an instruction), computed jumps make it impossible to reliably identify all instructions in untrusted binary code; disassemblers must heuristically guess the addresses of many instruction sequences to generate a complete disassembly. Untrusted binaries (e.g., malicious code) are often specifically crafted to defeat these heuristics, thereby concealing malicious instruction sequences from analysis tools.

2.2 System Overview

Given an untrusted binary, REINS automatically transforms it so that (1) all access to system (and library) APIs are mediated by our policy enforcement library, and (2) all inter-module control-flow

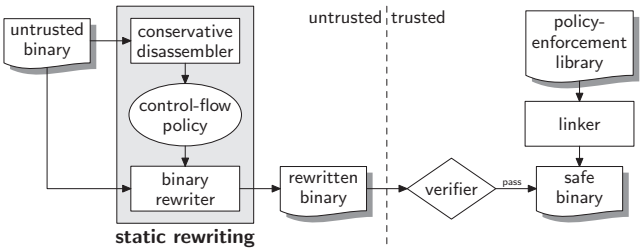

Figure 1: REINS architecture

transfers are restricted to published entry points of known libraries, preventing execution of attacker-injected or misaligned code.

REINS' rewriter first generates a conservative disassembly of the untrusted binary that identifies all safe, non-branching flows (some of which might not actually be reachable) but not unsafe ones. The resulting disassembly encodes a control-flow policy: instructions not appearing in the disassembly are prohibited as computed jump targets. Generating even this conservative disassembly of arbitrary x86 COTS binaries is challenging because COTS code is typically aggressively interleaved with data, and contains significant portions that are only reachable via computed jumps. To help overcome some of these challenges, our rewriter is implemented as an IDAPython [9] program that leverages the considerable analysis power of the Hex-rays IDA Pro commercial disassembler to identify function entrypoints and distinguish code from data in complex x86 binaries. While IDA Pro is powerful, it is not perfect; it suffers numerous significant disassembly errors for almost all production-level Windows binaries. Thus, our rewriting algorithm's tolerance of disassembly errors is critical for success.

Our system architecture is illustrated in Fig. 1. Untrusted binaries are first analyzed and transformed into safe binaries by a binary rewriter, which enforces control-flow safety and mediates all API calls. A separate verifier certifies that the 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. DETAILED DESIGN

3.1 Rewriting Control-flow Transfers

Control Flow Safety. Our binary rewriting algorithm uses SFI [36] to constrain control-flows of untrusted code. It is based on an SFI approach pioneered by PittSFIeld [20], which partitions 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 efficiently confined to chunk boundaries by guarding them with an instruction that dynamically clears the low-order bits of the jump target.
- Co-locating guards 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 sequence in our system.
- Aligning all code to c-byte boundaries allows a simple, fall-through disassembler to reliably discover all reachable instructions in rewritten programs, and verify that all computed jumps are suitably guarded.

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 and non-executable data. Higher memory addresses may contain code sections of trusted libraries and arbitrary data sections (but not untrusted code).

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

```
and eax, (d − c)
jmp eax
```

This clears both the high-order and low-order bits of the target address before jumping, preventing an untrusted module from jumping directly to a system accessor function or to a non-chunk boundary in its own code, respectively. The partitioning of virtual addresses into low and high memory is feasible 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.

Preserving Good Flows. The above suffices to enforce control-flow safety, but it does not preserve the behavior of most code containing computed jumps. This is a major deficiency of many early SFI works, most of which can only be successfully applied to relatively small, `gcc`-compiled programs that do not contain such jumps. More recent SFI works have only been able to overcome this problem with the aid of source-level debug information.

Our source-free solution capitalizes on the fact that although disassemblers cannot generally identify all jumps in arbitrary binary code, modern commercial disassemblers can heuristically identify a *superset* of all the indirect jump *targets* (though not the jumps that target them) in most binary code. This is enough information to implement a light-weight, binary lookup table that the IRM can consult at runtime to dynamically detect and correct computed jump targets before they are used. Our lookup table overwrites each old target with a tagged pointer to its new location in the rewritten code. This solves the computed jump preservation problem without the aid of source code.

Since the disassembler identifies a superset of targets and not an exact set, the lookup table implementation must be carefully designed to tolerate false positives. Misidentification of a code point as a jump target is therefore relatively harmless to REINS; each such misidentification merely increases the size of rewritten code by a few bytes due to alignment. A false negative (i.e., failure to identify one or more targets) is more serious and may lead to rewritten code that does not execute properly, but the verifier ensures that it cannot lead to a policy violation. Thus, both forms of error are tolerated.

Another major design issue is the need to arrange the lookup table so that IRM code that uses it remains exceptionally small and efficient. This is critical for achieving low overhead, since computed jumps are extremely common in real-world binaries. Our solution implements most lookups with just two non-branching instructions (a compare instruction and a conditional move), shown atop the first row of Table 1. This efficient implementation is achieved by tagging each lookup table entry with a leading byte that never appears 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. The compare instruction uses this byte to quickly distinguish stale pointers that point into the lookup table from those that already point to code. The conditional move then corrects the stale ones. This succinct realization of semantics-preserving computed jump guards is the key to REINS' exceptionally low overhead.

Retaining the old code section as a data section has the additional advantage of retaining any static data that may be interleaved in the

Table 1: Summary of x86 code transformations

Description	Original code	Rewritten code

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

Computed jumps with memory operands	`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`, [IAT:`n`]`	`mov` rm`, offset tramp_`n

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

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 computed jump employs a memory operand instead of a register, as shown in row 2 of Table 1, the rewritten code requires a scratch register. Table 1 uses `eax`, which is caller-save by convention and is not used to pass arguments by any calling convention supported by any mainstream x86 compiler [12].[1]

A particularly common form of computed jump deserves special note. Return instructions (`ret`) jump to the address stored atop the stack (and optionally pop n additional bytes from the stack afterward). 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 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 they block jumps to corrupted return addresses that point to illegal destinations, such as the stack. This makes all attacker-injected code unreachable.

Preserving API Calls. 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 *import address table* (IAT) are retained. Such jumps usually have the following form:

`call [IAT:`n`]`

where `IAT` is the section of the executable reserved for the IAT and n is an offset that identifies the IAT entry. These jumps are safe since the entrypoint to the APIs is hooked by REINS to ensure that they always target policy-compliant addresses at runtime.

Not all uses of the IAT have this simple form, however. Most x86-targeting compilers also generate optimized code that caches IAT entries to registers, and uses the registers as jump targets. 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 callee-specific *trampoline chunk* (in row 5) introduced to the rewritten code section (if it doesn't already exist). The trampoline chunk safely jumps to the trusted callee using a direct IAT reference. Thus, any use of the replacement pointer as a jump target results in a jump to the trampoline, which invokes the desired function.

Dynamic linking and callbacks are both supported via a similar form of trampolining detailed in the technical report [16].

[1]To support binaries that depend on preserving `eax` across computed jumps, the table's sequence can be extended with two instructions that save and restore `eax`. We did not encounter any programs that require this, so our experiments use the table's shorter sequence.

```
Original:
.text:00499345  8B 35 FC B5 4D 00     mov esi, [4DB5FCh] ;IAT:MBTWC
...
.text:00499366  FF D6                 call esi

Rewritten:
.tnew:0059DBF0  BE 90 12 5D 00        mov esi, offset loc_5D1290
...
.tnew:0059DC15  80 3E F4              cmp byte ptr [esi], F4h
.tnew:0059DC18  0F 44 76 01           cmovz esi, [esi+1]
.tnew:0059DC1C  90 90 90 90           nop (×4)
.tnew:0059DC20  81 E6 F0 FF FF 0F     and esi, 0FFFFFF0h
.tnew:0059DC26  90 (×8)               nop (×8)
.tnew:0059DC2E  FF D6                 call esi
...
.tnew:005D1290  81 24 24 F0 FF FF 0F  and dword ptr [esp], 0FFFFFF0h
.tnew:005D1297  FF 25 FC B5 4D 00     jmp [4DB5FCh] ;IAT:MBTWC
```

Figure 2: Rewriting a register-indirect system call

```
Original:
.text:00408495  FF 24 85 CC 8A 40 00  jmp ds:off_408ACC[eax*4]
...
.text:00408881  3D 8C 8A 4D 00 00     cmp byte_4D8A8C, 0
.text:00408888  74 13                 jz short loc_40889D
.text:0040888A  84 C9                 test cl, cl
.text:0040888C  74 0F                 jz short loc_40889D
...
.text:00408ACC  81 88 40 00           dd offset loc_408881
.text:00408AD0  ...                   (other code pointers)

Rewritten:
.text:00408881  F4 60 3A 4F 00        db F4, loc_4F3A60
                — — — — — — — — — — — — — — — —
.tnew:004F33B4  8B 04 85 CC 8A 40 00  mov eax, ds:dword_408ACC[eax*4]
.tnew:004F33BB  80 38 F4              cmp byte ptr [eax], F4h
.tnew:004F33BE  90 90                 nop (×2)
.tnew:004F33C0  0F 44 40 01           cmovz eax, [eax+1]
.tnew:004F33C4  25 F0 FF FF 0F        and eax, 0FFFFFF0h
.tnew:004F33C9  FF E0                 jmp eax
...
.tnew:004F3A60  3D 8C 8A 4D 00        cmp byte_4D8A8C, 0
.tnew:004F3A67  74 27                 jz short loc_4F3A90
.tnew:004F3A69  84 C9                 test cl, cl
.tnew:004F3A6B  74 22                 jz short loc_4F3A90
```

Figure 3: Rewriting code that uses a jump table

3.2 Examples

To illustrate the rewriting algorithm, Figs. 2 and 3 demonstrate the transformation process for two representative assembly codes.

Figure 2 implements a register-indirect call to a system API function (MBTWC). The first instruction of the original code loads an IAT entry into the `esi` register, which is later used as the target of the call. REINS replaces this address with the address of the in-lined trampoline code at the bottom of the figure, which performs a safe jump to the same destination. The call instruction is replaced with the guarded call sequence shown in lines 2–7 of the rewritten binary. The compare (`cmp`) and conditional move (`cmovz`) implement the table-lookup, and the masking instruction (`and`) aligns the destination to a chunk boundary. This makes the ensuing call provably safe to execute.

Figure 3 shows a computed jump with a memory operand that indexes a jump table. The rewritten code first loads the destination address into a scratch register (`eax`) in accordance with row 2 of Table 1. It then implements the same lookup and masking guards as in Fig. 2. This time the lookup has a significant effect—it discovers at runtime that the address drawn from the lookup table must be repointed to a new address. This preserves the behavior of the binary after rewriting despite the failure of the disassembler to discover and identify the jump table at rewrite-time.

3.3 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 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. Thus, attacker-injected shell code in the stack or other data memory regions cannot be executed.

User processes on Windows systems can set or unset 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—e.g., `VirtualProtect` and `VirtualAlloc`. The rewriter replaces the IAT entries of these functions with trusted wrapper functions that 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 system API directly with modified arguments.

The real system functions are accessible to trusted libraries (but not untrusted libraries) because they have separate IATs that are not subjected to our IAT hooking. Trusted libraries can therefore use them to protect their local heap and stack pages from untrusted code that executes in the same address space. Our API hooks prevent rewritten code from directly accessing the page protection bits 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 a mainstay of certain application domains, such as JIT-compilers. For these domains we consider alternative technologies, such as certifying compilers and certified, bytecode-level IRMs, to be a more appropriate means of protection. Self-modifying code is increasingly rare in other domains, such as application installers, because it is incompatible with DEP, incurs a high performance penalty, and tends to trigger conservative rejection by antivirus products. No SFI system to our knowledge supports arbitrary self-modifying code.

3.4 Verification

The disassembler, rewriter, and lookup table logic all remain completely untrusted by our architecture. Instead, a small, independent verifier certifies that rewritten programs cannot circumvent the IAT and are therefore policy-adherent. The verifier does not prove that the rewriting process is behavior-preserving. This reduced obligation greatly simplifies the verifier relative to the rewriter, resulting in a small TCB.

The verification algorithm performs a simple fall-through disassembly of each executable section in the untrusted binary and checks the following purely syntactic properties:

- All executable sections reside in low memory.
- All exported symbols (including the program entrypoint) target low memory chunk boundaries.
- No disassembled instruction spans a chunk boundary.
- Static branches target low memory chunk boundaries.
- All computed jump instructions that do not reference the IAT are immediately preceded by the appropriate `and`-masking instruction from Table 1 in the same chunk.
- Computed jumps that read the IAT access a properly aligned IAT entry, and are preceded by an `and`-mask of the return address. (Call instructions must *end* on a chunk boundary rather than requiring a mask, since they push their own return addresses.)
- There are no trap instructions (e.g., `int` or `syscall`).

These properties ensure that any unaligned instruction sequences concealed within untrusted, executable sections are not reachable at runtime. This allows the verifier to limit its attention to a fall-through disassembly of executable sections, avoiding any reliance upon the incomplete code-discovery heuristics needed to produce full disassemblies of arbitrary (non-chunk-aligned) binaries.

4. IMPLEMENTATION

We have developed an implementation of REINS for the 32-bit version of Microsoft Windows XP/Vista/7/8. The implementation consists of four components: (1) a rewriter, (2) a verifier, (3) an API hooking utility, and (4) an intermediary library that handles dynamic linking and callbacks. Rather than using a single, static API hooking utility, we implemented an automated *monitor synthesizer* that generates API hooks and wrappers from a declarative policy specification. This is discussed in §5.3. None of the components require elevated privileges. While the implementation is Windows-specific, we believe the general approach is applicable to any modern OS that supports DEP technology.

The rewriter transforms Windows Portable Executable (PE) files in accordance with the algorithm in §3. Its implementation consists of about 1,300 lines of IDA Python scripting code that executes atop the Hex-rays IDA Pro 6.1 disassembler. One of IDA Pro's primary uses is as a malware reverse engineering and de-obfuscating tool, and 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 rewriting.

In contrast to the significant complexity of the rewriting infrastructure, the verifier's implementation consists of 1,500 lines of 80-column OCaml code that uses no external libraries or utilities (other than the built-in OCaml standard libraries). Of these 1,500 lines, approximately 1,000 are devoted to x86 instruction decoding, 300 to PE binary parsing, and 200 to the actual verification algorithm in §3.4. The decoder handles the entire x86 instruction set, including floating point, MMX, and all SSE extensions documented in the Intel and AMD manuals. This is necessary for practical testing since production-level binaries frequently contain at least some exotic instructions. No code is shared between the verifier and rewriter.

The intermediary library consists of approximately 500 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. 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 in exported function prototypes.

Our API hooking utility replaces the IAT entries of all monitored system functions imported by rewritten PE files with the addresses of trusted monitor functions. It also adds the intermediary library to the PE's list of imported modules. To avoid expanding the size of the PE header (which could shift the positions of the binary sections that follow it), our utility simply changes the library name `kernel32.dll` in the import section to the name of our intermediary library. This causes the system loader to draw all IAT entries previously imported from `kernel32.dll` from the intermediary library instead. The intermediary library exports all `kernel32` symbols as forwards to the real `kernel32`, except for security-relevant functions, which it exports as local replacements. Our intermediary library thus doubles as the policy enforcement library.

Table 2: Experimental results: SPEC benchmarks

Binary Program	Size Increase File (%)	Code (%)	Process (%)	Rewriting Time (s)	Verification Time (ms)
gzip	103	31	0	12.5	142
vpr	94	26	22	14.4	168
mcf	108	32	2	10.5	84
parser	108	34	1	17.4	94
gap	118	42	0	31.2	245
bzip2	102	29	0	10.8	91
twolf	99	24	27	25.3	245
mesa	104	20	6	42.4	554
art	108	33	14	12.4	145
equake	103	27	1	12.3	165
median	+103.5%	+30.0%	+1.5%	13.45s	155ms

Table 3: Experimental results: Applications and malware

Binary Program	Size Increase File (%)	Code (%)	Process (%)	Rewriting Time (s)	Verification Time (ms)
notepad	60	31	20	1.5	18
Eureka	32	53	15	17.9	225
DOSBox	112	38	0	137.1	2394
PhotoView	87	57	4	3.5	49
BezRender	128	55	3	4.1	55
gcc	100	37	15	3.0	36
g++	100	41	16	3.0	37
jar	101	34	12	2.4	27
objcopy	122	49	23	26.9	354
size	103	50	116	16.3	20
strings	122	50	42	21.5	283
as	99	49	2	30.4	397
ar	121	50	4	21.8	285
whetstone	88	21	54	0.6	6
linpack	57	19)	31	0.6	6
pi_ccs5	125	28	1	5.8	66
md5	25	48	149	0.6	5
median	100%	41%	15%	4.1s	49ms
Virut.a		(rejected)		–	–
Hidrag.a		(rejected)		–	–
Vesic.a	75	34	108	0.3	194
Sinn.1396	37	115	93	0.2	75
Spreder.a	14	66	17	3.0	72
median	37%	66%	93%	0.3s	75ms

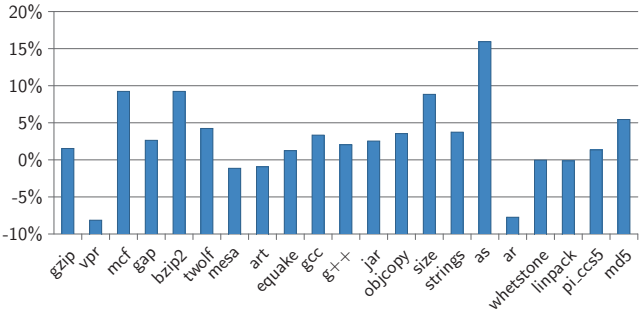

Figure 4: Runtime overhead due to rewriting

5. EVALUATION

5.1 Rewriting Effectiveness

We tested REINS with a set of binary programs listed in Tables 2 and 3. Table 2 lists results for some of the benchmarks from the SPEC 2000 benchmark suite. Table 3 lists results for some other applications, including GUI programs that include event- and callback-driven code, and malware samples that require enforcement of higher-level security policies to prevent malicious behavior. In both tables, columns 2–3 report the percentage increase of the file size, code segment, and process size, respectively; and columns 5–6 report the time taken for rewriting and verification, respectively. All experiments were performed on a 3.4GHz quad-processor AMD Phenom II X4 965 with 4GB of memory running Windows XP Professional and MinGW 5.1.6.

File sizes double on average after rewriting for benign applications, while malware shows a smaller increase of about 40%. Code segment sizes increase by a bit less than half for benign applications, and a bit more than half for malware. Process sizes typically increase by about 15% for benign applications, but almost 90% for malware. The rewriting speed is about 32s per megabyte of code, while verification is much faster—taking only about 0.4s per megabyte of code on average.

5.2 Performance Overhead

We also measured the performance of the non-interactive programs in Tables 2 and 3. The runtime overheads of the rewritten programs as a percentage of the runtimes of the originals is presented in Fig. 4. The median overhead is 2.4%, and the maximum is approximately 15%. As with other similar works [1, 13], the runtimes of a few programs decrease after rewriting. This effect is primarily due to improved instruction alignment introduced by the rewriting algorithm, which improves the effectiveness of instruction look-ahead and decoding pipelining optimizations implemented by modern processors. While the net effect is marginal, it is enough to offset the overhead introduced by the rest of the protection system in these cases, resulting in safe binaries whose runtimes are as fast as or faster than the originals.

The experiments reported in Tables 2 and 3 enforced only the core access control policies required to prevent control-flow and memory safety violations. Case studies that showcase the framework's capacity to enforce more useful policies are described in §5.4.

5.3 Policy Enforcement Library Synthesis

To quickly and easily demonstrate the framework's effectiveness for enforcing a wide class of safety policies, we developed a monitor synthesizer that automatically synthesizes the policy enforcement portion of the intermediary library from a declarative policy specification. Policy specifications consist of: (1) the module names and signatures of all security-relevant API functions to be monitored, (2) a description of the runtime argument values that, when passed to these API functions, constitute a security-relevant *event*, and (3) a regular expression over this alphabet of events whose prefix-closure is the language of permissible *traces* (i.e., event sequences).

To illustrate, Fig. 5 shows a sample policy. Lines 1–5 are signatures of two API functions exported by Windows system libraries: one for connecting to the network and one for creating files. Lines 7–8 identify network-connects as security-relevant when the outgoing port number is 25 (i.e., an SMTP email connection) and the return value is 0 (i.e., the operation was successful), and file-creations as security-relevant when the filename's extension is `.exe`. Underscores denote arguments whose values are not security-relevant. Finally, line 10 defines traces that include at most one kind of event (but not both) as permissible. Here, $\star$ denotes finite or infinite repetition and + denotes regular alternation.

Currently our synthesizer implementation supports dynamic value tests that include string wildcard matching, integer equality and inequality tests, and conjunctions of these tests on fields within a structure. From this specification, the monitor synthesizer generates the C source code of a policy enforcement library that uses IAT hooking to reroute calls to `connect` and `CreateFileW` through trusted guard functions. The guard functions implement the desired

```
1  function conn = ws2_32::connect(
2    SOCKET, struct sockaddr_in *, int) -> int;
3  function cfile = kernel32::CreateFileW(
4    LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
5    DWORD, DWORD, HANDLE) -> HANDLE WINAPI;

7  event e1 = conn(_, {sin_port=25}, _) -> 0;
8  event e2 = cfile("*.exe", _, _, _, _, _, _) -> _;

10 policy = e1* + e2*;
```

Figure 5: A policy that prohibits applications from both sending emails and creating `.exe` files

```
1  function cfile = kernel32::CreateFileW(
2    LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
3    DWORD, DWORD, HANDLE) -> HANDLE WINAPI;
4  function exec = kernel32::WinExec(LPCSTR, UINT)
5    -> UINT WINAPI;

7  event e1 = cfile("*.exe", _, _, _, _, _, _) -> _;
8  event e2 = cfile("*.msi", _, _, _, _, _, _) -> _;
9  event e3 = cfile("*.bat", _, _, _, _, _, _) -> _;
10 event e4 = exec("explorer", _) -> _;

12 policy = ;
```

Figure 6: Eureka email policy

policy as a determinized *security automaton* [30]—a finite state automaton that accepts the prefix-closure of the policy language in line 10. If the untrusted code attempts to exhibit a prohibited trace, the monitor rejects by halting the process.

5.4 Case Studies

5.4.1 An Email Client

As a more in-depth case-study, we used the rewriting system and monitor synthesizer to enforce two policies on the *Eureka* 2.2q email client. Eureka is a fully featured, commercial POP client for 32-bit Windows that features a graphical user interface, email filtering, and support for launching executable attachments as separate processes. It is 1.61MB in size and includes all of the binary features discussed in earlier sections, including Windows event callbacks and dynamic linking. It statically links to eight trusted system libraries.

Without manual assistance, IDA automatically recovers enough structural information from the Eureka binary to facilitate the full binary rewriting algorithm presented in §3. Rewriting requires 18s and automated verification of the rewritten binary requires 0.2s.

After rewriting, we synthesized an intermediary library that enforces the access control policy given in Fig. 6, which prohibits creation of files whose filename extensions are `.exe`, `.msi`, or `.bat`, and which prevents the application from launching Windows Explorer as an external process. (The empty policy expression in line 12 prohibits all events defined in the specification.) We also enforced the policy in Fig. 5, but with a policy expression that limits clients to at most 100 outgoing SMTP connections per run. Such a policy might be used to protect against malware infections that hijack email applications for propagation and spamming.

After rewriting, we systematically tested all program features and could not detect any performance degradation or changes to any policy-permitted behaviors. All program features unrelated to the policy remain functional. However, saving or launching an email attachment with any of the policy-prohibited filename extensions causes immediate termination of the program by the monitor. Likewise, using any program operation that attempts to open an attachment using Windows Explorer, or sending more than 100 email messages, terminates the process. The rewritten binary therefore correctly enforces the desired policy without impairing any of the application's other features.

5.4.2 An Emulator

DOSBox is a large DOS emulator with over 16 million downloads on sourceforge. Though its source code is available, it was not used during the experiment. The precompiled binary is 3.6MB, and like Eureka, includes all the difficult binary features discussed earlier.

We enforced several policies that prohibit access to portions of the file system based on filename string and access mode. We then used the rewritten emulator to install and use several DOS applications, including the games Street Fighter 2 and Capture the Flag. Installation of these applications requires considerable processing time, and is the basis for the timing statistics reported in Table 3. As in the previous experiment, no performance degradation or behavioral changes are observable in the rewritten application, except that policy-violating behaviors are correctly prohibited.

5.4.3 Malware

To analyze the framework's treatment of real-world malware, we tested REINS on five malware samples obtained from a public malware research repository: *Virut.a*, *Hidrag.a*, *Vesic.a*, *Sinn.1396*, and *Spreder.a*. While these malware variants are well-known and therefore preventable by conventional signature-matching antivirus defenses, the results indicate how our system reacts to binaries intentionally crafted to defeat disassembly tools and other static analyses. Each is statically or dynamically rejected by the protection system at various different stages, detailed below.

Virut and *Hidrag* are both rejected at rewriting time when the rewriter encounters misaligned static branches that target the interior of another instruction. While supporting instruction aliasing due to misaligned *computed* jumps is useful for tolerating disassembly errors, misaligned *static* jumps only appear in obfuscated malware to our knowledge, and are therefore conservatively rejected.

Vesic and *Sinn* are Win32 viruses that propagate by appending themselves to executable files on the C: volume. They do not use packing or obfuscation, making them good candidates for testing our framework's ability to detect malicious behavior rather than just suspicious binary syntax. With a fully permissive policy, our framework successfully rewrites and verifies both malware binaries; running the rewritten binaries preserves their original (malicious) behaviors. However, enforcing the policy in Fig. 6 results in premature termination of infected processes when they attempt to propagate by writing to executable files. We also successfully enforced a second policy that prohibits the creation of system registry keys, which *Vesic* uses to insert itself into the boot process of the system. These effectively protect the infected system before any damage results.

Spreder has a slightly different propagation strategy that searches for executable files in the shared directory of the Kazaa file-sharing peer-to-peer client. We successfully enforced a policy that prohibits use of the `FindFirstFileA` system API function to search for executable files in this location. This results in immediate termination of infected processes.

6. DISCUSSION

In this section we first discuss the security benefits REINS provides, and then discuss the binary code conventions that are prerequisites for behavior-preservation under our binary rewriting scheme, as well as the reliability of our disassembly. The limitations of our approach are highlighted during the course of the discussion.

6.1 Control-flow Policies

As we have demonstrated, REINS can rewrite many complex legacy binaries, enforcing coarse-grained control-flow safety and preserving safe computed jumps without source code. However, REINS does not enforce the finer-grained control-flow integrity properties of CFI [1]. CFI uses source code or PDB files to build a control-flow graph that serves as the integrity policy to enforce. This connection to source code is foundational to CFI because any fine-grained definition of "good" control-flows invariably depends on the semantics of the source code that the untrusted binary code is intended to reflect. Without source code, there is no sensible definition of control-flow integrity for REINS to enforce.

As such, REINS and CFI have fundamentally different goals. CFI's goal is to micro-manage behavior within an untrusted binary to prevent attackers from corrupting its internal flows. In contrast, REINS' goal is to protect the environment outside the untrusted binary, not its internals. This includes external resources like the file system and network, and the trusted libraries that access them (e.g., OS/kernel libraries). The only flows that affect such resources are those that exit the untrusted code. For these, there are sensible, well-defined (but coarser-grained) control-flow policies apart from source code. For example, flows to the stack or data are disallowed (to block code-injection attacks), and flows to trusted libraries must obey the library's interface (e.g., its export address table).

REINS prevents these forms of malicious behavior based on the security policy. It is possible for an attacker to craft ROP [33] or Q [31] shell code to overwrite the stack pointer and break the internal control flows, but the attacker must ultimately manipulate the arguments of system calls to effect damage outside the confines of the untrusted module. These malicious system calls are detected and prevented by REINS.

6.2 Code Conventions

Our rewriting algorithm in §3 preserves the behavior of code that adheres to standard, compiler-agnostic x86 code generation conventions. Code that violates these conventions can yield rewritten code that fails verification or fails to execute properly, but never verified code that circumvents the monitor. 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.

Code pointers. REINS expects each code pointer used as a jump target by untrusted code to originate from one of five sources:

- a low-memory address drawn from the program counter (e.g., a return address pushed by a `call`),
- data that points to a basic block boundary,
- a code address 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.

As demonstrated by our experiments, these cover a large spectrum of real-world binary code. Nevertheless, there are some unusual cases that REINS still rejects. For example, a program that computes external library entrypoints instead of requesting them from the system's linker is incompatible with REINS, and will typically crash when executed. Addressing such limitations is future work.

Reliable Disassembly. Binaries generated by most mainstream compilers mix code and static data within the `.text` section of the executable. REINS relies upon a classification algorithm that heuristically distinguishes code from data [38]. If code is misclassified as data, that code is incorrectly omitted from the rewritten binary's code section. If data is misclassified as code that looks like a possible computed jump target, the rewriter might overwrite some of the data with tagged pointers as it constructs the lookup table (see §3.1). This can result in corruption of the static data. However, data misclassified as code without such targets just contributes harmless, dead code to the rewritten binary's code section. Heuristics that conservatively classify most bytes as code with few computed jump targets therefore tend to work well for our system.

Function entrypoints 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 the disassembler's code reachability analysis and a few 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 found that for most non-malicious programs, IDA Pro's automatic binary analysis works well, accurately identifying all code (with some data harmlessly misidentified as code) and identifying all computed jump targets (with some code harmlessly misidentified as a computed jump target). Any missed targets are easy to identify and correct manually, since their omission causes the rewritten binary to crash at precisely the site of the misclassified address in the (now non-executable) old code segment.

Dense Computed Jump Targets. A more subtle assumption of the algorithm is that all computed jump targets in the original binary are at least $w + 1$ bytes away from the next computed jump target or following data, where w is the system word size. This is necessary to ensure sufficient space for the rewriter to write a tagged pointer at that address without overwriting any adjacent pointers or data. Entrypoints packed closer than this are rare, since most computed jump targets are 16-byte aligned for performance reasons, and since all binaries compatible with *hotpatching* have at least $w + 1$ bytes of padding between consecutive function entrypoints [21].

In the rare case that two targets are within w bytes in the original code, the rewriter strategically chooses the address of the rewritten code section so that the encodings of tagged pointers into it can occasionally overlap. For example, with tag byte $t =$ `0xF4`, the sequence `F4 00 F4 00 04 00 04` encodes two overlapping, little-endian tagged pointers to addresses `0x0400F400` and `0x04000400`. By positioning the rewritten versions of these two functions at those addresses, the rewriter can encode overlapping pointers to them in the lookup table. With chunk size $c = 16$ and memory division $d = 2^{28}$, a rewritten code base address of $2^{24}(t \,\&\, 0\text{xF}) + 2^{16}t$ supports at least 15 two-pointer collisions and 1 three-pointer collision per rewritten code page—far more than we saw in any binary we studied.

6.3 Other Future Work

The experiments reported in §5 focus on testing the soundness, transparency, and feasibility of our static binary rewriting algorithm on a real-world OS, and on demonstrating the enforcement of some simple but useful security policies. Past work [17, 19, 28] has shown that IRM systems are capable of enforcing more sophisticated temporal properties when equipped with more powerful event languages and responses to impending policy violations that go beyond mere program termination. 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.

7. RELATED WORK

REINS is related to SFI, whose works can be divided into (1) source-level approaches, which instrument untrusted code with dynamic security guards at compile-time, (2) binary-level approaches,

which secure untrusted code at a purely binary level, and (3) system-level approaches, which secure the software at system call level. Table 4 summarizes and compares the major feature differences of the related works mentioned below.

Source-level Approaches. Most SFI implementations target source code and therefore insert security guard instructions at compile-time. Examples include StackGuard [8], DFI [5], WIT [2], BGI [6], G-Free [23], and CFL [4]. Source-level approaches differ significantly from the problem of securing COTS native code because a compiler typically has full control over the structure of the binary it generates, its pointer representations, and its implementation of computed jumps. In contrast, SFI systems for legacy native code cannot statically distinguish code pointers from data, recover control-flow or data-flow graphs reliably, or detect all instruction aliasing. Enforcing SFI without this information introduces many challenges.

The primary disadvantage of source-level approaches is their reliance on the support of a cooperating code-producer, who must (re)compile the untrusted or insecure code using a special compiler. Such cooperation is not a reasonable expectation for many classes of untrusted code, which are distributed as raw native code produced by arbitrary compilers, and that target mainstream system APIs such as Microsoft Windows or Linux.

Binary-level Approaches. In contrast, binary-level approaches require less compiler cooperation. They can be further divided into those that operate dynamically and those that operate statically:

Dynamic binary approaches use dynamic binary translation (e.g., Vx32 [13], Strata [32], Libdetox [25]), program shepherding [18, 3], or safe loading (e.g., TRuE [26]) to dynamically copy and instrument untrusted code into a sandbox at runtime. Any flows that attempt to escape the sandbox recursively re-trigger the copying process, keeping all untrusted, reachable code within the sandbox.

In contrast, static binary approaches in-line guard instructions into untrusted binary code statically before the code executes, and do not perform any code generation or translation at runtime. The only SFI systems other than REINS that target legacy, untyped, native code binaries to our knowledge are CFI [1]/XFI [10], PittSFIeld [20], NaCl [39], and SecondWrite [34]. CFI/XFI achieves reliable disassembly by consulting PDB files, which contain debugging information. The debugging information reveals important structural and typing information from the application source code without disclosing the source code text; however, PDB files are only produced by Microsoft compilers, and most code-producers do not disclose them to the public. This significantly limits the domain of binaries to which CFI/XFI is applicable. PittSFIeld and NaCl are similarly limited—PittSFIeld only supports `gcc`-produced assembly code and NaCl requires untrusted code to be (re)compiled by their tool chain.

SecondWrite tackles the problem of rewriting COTS binaries without debug or relocation metadata, but it does not support formal machine-verification, has not yet been applied to realize complete fault isolation, and is not yet mature enough to rewrite full-scale COTS applications [24].

Dynamic vs. static approaches have historically suffered a compatibility vs. performance trade-off. That is, the dynamic approaches can currently handle a much larger class of binaries than the static ones, including large-scale COTS applications, but at the expense of significant runtime overheads (e.g., 70% slowdown in Strata [32]). In addition, dynamic SFI systems are difficult to formally verify, and cannot be deployed on architectures that prohibit runtime code generation or that lack the hardware-level VM support that is often necessary to achieve reasonable performance.

In contrast, static approaches offer much lower overheads and formal, machine-checkable proofs of safety, but currently support only a very restricted set of binary programs that do not include most COTS applications. Therefore, REINS is the first purely static binary SFI system capable of supporting nearly arbitrary, large-scale, COTS Windows applications produced by mainstream compilers, including those that contain computed jumps, dynamic linking, and event-driven OS callbacks.

Table 4: Summary of related works. Symbol $*$ represents a limited feature for that work.

System	no source needed	no metadata needed	supports COTS binaries	handles computed jumps	handles callbacks	non-system approach	no kernel privileges	automated verification	stack exploit protection	system call interposition	enforces API policies	control-flow integrity	source available
StackGuard [8]						✓	✓		✓				✓
DFI [5]				✓	*	✓	✓		✓				
WIT [2]				✓		✓	✓		✓			✓	
BGI [6]				✓	✓				✓			✓	
G-Free [23]				✓		✓	✓		*				
CFL [4]				✓		✓	✓		*			*	
Vx32 [13]	✓	✓	✓	✓	✓	✓	✓			✓	✓		✓
Strata [32]	✓	✓	✓	✓	✓	✓	✓			✓	✓		
Libdetox [25]	✓	✓	✓	✓	✓	✓	✓		*	✓	✓		✓
Shepherd [18]	✓	✓	✓	✓	✓	✓	✓		*	✓	✓		
TRuE [26]	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓		✓
CFI [1]	✓		*	*	*	✓	✓		✓	✓	*	✓	
XFI [10]	✓		*	*	*	✓	✓	✓	✓	✓	✓	✓	
PittSFIeld [20]	*			*		✓	✓	✓	✓	✓			✓
NaCl [39]	*			*		✓	✓	✓	✓	✓	✓		✓
2ndWrite [24]	✓	✓	*	✓		✓	✓		✓				
REINS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		[29]
Janus [35]	✓	✓	✓				✓			✓	*		✓
SysTrace [27]	✓	✓	✓							✓	*		✓
Ostia [15]	✓	✓	✓							✓	*		

System-level Approaches. There are also many system-level approaches, such as Janus [35], SysTrace [27], and Ostia [15]. These use system call interposition to enforce policies that prevent abuse of the system API.

Unlike binary rewriting approaches, system-level approaches are transparent to the binary code. However, they cannot block attacks of one module upon another within the same address space, they cannot be deployed as a service (because the full implementation must reside on the client machine), and they can introduce compatibility problems, such as incorrect replication of OS semantics [14].

8. CONCLUSION

We have presented the design, implementation, and evaluation of a new SFI/IRM system, REINS, that monitors and restricts Windows API calls of untrusted native x86 binaries for which source code and debugging information are unavailable. The binary rewriting algorithm supports many difficult binary features, including computed jumps, dynamic linking, interleaved code and data, and OS callbacks, all without any explicit cooperation from code-producers, and it is behavior-preserving for a large class of COTS binaries. To the best of our knowledge, no past binary rewriting-based SFI work has achieved this. The enforcement mechanism requires no kernel extensions or privileges, making it applicable to shared and closed computing environments, and separate, light-weight machine-verification keeps the TCB small. Experiments on a number of COTS and malware programs show the effectiveness of REINS, and demonstrate that although rewriting doubles file sizes on average, runtimes increase by only about 2.4% and the median process size increases by only about 15%.

9. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Information and System Security*, 13(1), 2009.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. IEEE Sym. Security and Privacy*, pages 263–277, 2008.

[3] P. Bania. Securing the kernel via static binary rewriting and program shepherding. http://piotrbania.com/all/articles/pbania-securing-the-kernel2011.pdf, 2011.

[4] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proc. Annual Computer Security Applications Conf.*, pages 353–362, 2011.

[5] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proc. USENIX Sym. Operating Systems Design and Implementation*, pages 147–160, 2006.

[6] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proc. ACM Sym. Operating Systems Principles*, pages 45–58, 2009.

[7] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proc. Workshop Hot Topics in Operating Systems*, pages 133–138, 2001.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. USENIX Security Sym.*, 1998.

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

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

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

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

[13] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proc. USENIX Annual Technical Conf.*, pages 293–306, 2008.

[14] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proc. Network and Distributed System Security Sym.*, 2003.

[15] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Sym.*, 2004.

[16] K. W. Hamlen, V. Mohan, and R. Wartell. Reining in Windows API abuses with in-lined reference monitors. Technical Report UTDCS-18-10, U. Texas at Dallas, 2010.

[17] 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.

[18] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. USENIX Security Sym.*, pages 191–206, 2002.

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

[20] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. USENIX Security Sym.*, 2006.

[21] Microsoft Corporation. Using hotpatching technology to reduce servicing reboots. *TechNet Library*, 2005. http://technet.microsoft.com/en-us/library/cc787843.aspx.

[22] G. C. Necula. Proof-carrying code. In *Proc. ACM Principles of Programming Languages*, pages 106–119, 1997.

[23] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proc. Annual Computer Security Applications Conf.*, pages 49–58, 2010.

[24] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in COTS software with binary rewriting. In *Proc. Int. Information Security Conf.*, pages 154–172, 2011.

[25] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, pages 157–168, 2011.

[26] M. Payer, T. Hartmann, and T. R. Gross. Safe loading – a foundation for secure execution of untrusted programs. In *Proc. IEEE Sym. Security and Privacy*, pages 18–32, 2012.

[27] N. Provos. Improving host security with system call policies. In *Proc. USENIX Security Sym.*, 2003.

[28] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting. System call monitoring using authenticated system calls. *IEEE Trans. Dependable and Secure Computing*, 3(3):216–229, 2006.

[29] Reins source code. http://sourceforge.net/projects/x86reins.

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

[31] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proc. USENIX Security Sym.*, 2011.

[32] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proc. Annual Computer Security Applications Conf.*, pages 209–218, 2002.

[33] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM Conf. Computer and Communications Security*, pages 552–561, 2007.

[34] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua. Binary rewriting without relocation information. Technical report, U. Maryland, 2010.

[35] D. A. Wagner. Janus: An approach for confinement of untrusted applications. Master's thesis, U. California at Berkeley, 1999.

[36] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. ACM Sym. Operating Systems Principles*, pages 203–216, 1993.

[37] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. ACM Conf. Computer and Communications Security*, 2012. in press.

[38] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham. Differentiating code from data in x86 binaries. In *Proc. European Conf. Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, volume 3, pages 522–536, 2011.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. IEEE Sym. Security and Privacy*, pages 79–93, 2009.