REWRITING X86 BINARIES WITHOUT CODE PRODUCER COOPERATION

by

Richard Wartell

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Kevin Hamlen, Chair

Dr. Gopal Gupta

Dr. Murat Kantarcioglu

Dr. Zhiqiang Lin

Dedicated to my parents,

for their unending support.

And my brother,

for his inspiration and honesty.

REWRITING X86 BINARIES WITHOUT CODE PRODUCER COOPERATION

by

RICHARD WARTELL, B.S.

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2012

ACKNOWLEDGMENTS

The author is incredibly grateful for the opportunity to work under his advisor, Dr. Kevin Hamlen, who was thoroughly helpful in building the foundation of the author's knowledge necessary to complete this dissertation and inspiring a thirst for knowledge and creativity. His patient and humble approach to teaching and direction were crucial in the author's research and the completion of this work.

This research would never have been accomplished without the help of the author's colleague, Vishwath Mohan. His willingness to act as a sounding board for ideas and helpfulness in working on certain pieces of this work were crucial in its completion. Special thanks should also be given to Dr. Yan Zhou whose work in Machine Learning and collaboration on disassembly projects was unendingly helpful to the author. Special mention should also be given to Dr. Zhiqiang Lin for his contributions and ideas that extended the author's research. Meera Sridhar should also be thanked for her work ethic and contribution to the work on rewriter transparency. Finally, thanks should also be given to the work done by Dr. Micah Jones and Dr. Murat Kantarcioglu whose work is closely tied to this dissertation.

This research was supported in part by Air Force Office of Scientific Research awards FA9550-08-1-0044 and FA9550-10-1-0088, and National Science Foundation award #1054629. All opinions expressed are those of the authors and do not necessarily reflect those of the AFOSR or NSF.

The author extends his heartfelt thanks Dr. Mikhail Atallah who was the original inspiration for this author's interest in security. Additionally, the author would like to thank Matt White who was this author's original collaborator, inspiration, and good friend. Finally, the author wishes to thank Nate Gatchell, whose friendship, support, and unending spirit and swagger were a constant inspiration.

November 2012

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the “Guide for the Preparation of Master’s Theses and Doctoral Dissertations at The University of Texas at Dallas.” It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student’s contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

REWRITING X86 BINARIES WITHOUT

CODE PRODUCER COOPERATION

Publication No. ____________________

Richard Wartell, Ph.D.
The University of Texas at Dallas, 2012

Supervising Professor: Dr. Kevin Hamlen

Binary code from untrusted sources remains one of the primary vehicles for software propagation and malicious software attacks. All previous work to mitigate such attacks requires code-producer cooperation, has significant deployment issues, or incurs a high performance penalty. The problem of accurate static x86 disassembly without metadata is provably undecidable, and is regarded by many as uncircumventable.

This dissertation presents a framework for x86 binary rewriting that 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 framework is instantiated as two software security systems: STIR, a runtime basic block randomization rewriter for Return-oriented programming (ROP) attack mitigation, and REINS, a machine verifiable Software Fault Isolation (SFI) and security policy specification rewriter. Both systems exhibit extremely low performance overheads in experiments on real-world COTS software— 1.6%

and 2.4% respectively. The foundation of the system includes three novel approaches to static x86 disassembly, along with a method of statically proving transparency for rewriting systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Native code is presently the most ubiquitous and prevalent form of mobile software. In 2008 alone, more than 288 million PCs were shipped, with a combined 3.6 billion units of software shipped as well [Business Software Alliance, 2010]. The majority of individuals and organizations do not have the money or manpower to produce all their own software, and therefore must blindly trust a large volume of native code obtained from untrusted or semi-trusted sources.

The software business has been growing ever since its inception. In 2008 alone, 3.6 billion packaged software units were sold worldwide, with an estimated total revenue of $296 billion dollars [Business Software Alliance, 2010]. In 2010, the estimated revenue of enterprise software alone (i.e., software purchased by organizations not individuals) was $244 billion dollars [Gartner, 2010]. These numbers only represent mainstream software purchases and downloads. They do not include software downloads of freeware, shareware, and open-source projects, which are harder to estimate.

Untrustworthy code sources are not limited to malicious software authors; in security-critical contexts they also include any source that is fallible. Faulty software is typically vulnerable software that can be subverted by an adversary to become malicious software. As a result, formal verification of a software's origin (e.g., by digital signature verification) provides few tangible security assurances. In fact, most software is a conglomeration of many components from many different vendors spread across many countries worldwide. A flaw in any one of these components can often be exploited to subvert the entire software and the entire system on which it runs. Mitre's Common Vulnerabilities and Exposures (CVE) database has catalogued over 5,500 highest-severity software vulnerabilities (i.e., those that lead to complete system compromise) in major software products just since 2010

The software engineering, programming languages, and compiler design communities have a long and active history of discovering ever more powerful tools and practices for the design and implementation of secure software that is free from such faults. The Coq automated theorem proving language [INRIA, 2012] is the result of 20 years of research by the programming language community, allowing formal machine verifiable proofs to be provided with software developed entirely within the Coq framework. The software engineering community is increasingly focused on security requirements engineering, attempting to mitigate software faults via improved programming practices. However, it is unlikely that 100% of software will be developed using these technologies in the near future. The vast majority of the industry continues to develop their products in unsafe languages, such as C++, for which formal proofs of security are intractable or impossible. Such practices show little evidence of abatement.

In response to the inevitability of untrusted native code, code-consumers have historically adopted three major approaches to safely executing untrusted or untrustworthy native code applications:

1. Many security-conscious organizations, including many governments, develop most or all of their security-critical software in-house using secure tools and a detailed code review process.

2. Some untrusted software can be safely executed as-is within a sandboxing virtual machine (VM).

3. More recently, technologies for secure binary analysis and transformation have emerged that promise to safely filter untrustworthy code or statically transform it into safe code.

Developing in-house software may seem like the best option, but it can be a slow, time and resource consuming process. In addition, it can be just as prone to error as mainstream development, even given cutting-edge tools. Most code consumers—even governments—lack resources or personnel comparable to the worldwide software development community, which includes industry titans like Microsoft, Google, and Apple. In-house development therefore constantly lags well behind the state-of-the-art, remaining susceptible to attacks that the rest of the world has addressed long ago.

Executing software within a VM [VMware, 2012] is a viable solution, protecting a host machine from faulty software run within the VM. VMs also offer the opportunity to run guest operating systems within a host machine—a much more reasonable solution than owning a computer that uses each operating system. However, though VMs provide numerous benefits, there are many situations they cannot handle:

1. VMs often cannot tractably enforce fine-grained policies. For example, VMware does not support enforcement of fault isolation for individual modules within process address spaces, because doing so requires a knowledge of the internal process layout and code logic that is beyond VMware's capabilities.

2. Enforcing a new policy requires modifying the VM, which tends to be very difficult and error-prone.

3. VM-sandboxed processes cannot easily access external system resources like trusted processes on the host machine without severe performance penalties. This makes them unsuitable for many applications.

4. VMs are hard to prove correct; they must usually remain trusted, and they're very complex and subject to frequent change. The size of the free version of VMware Workstation installation file is currently 426MB.

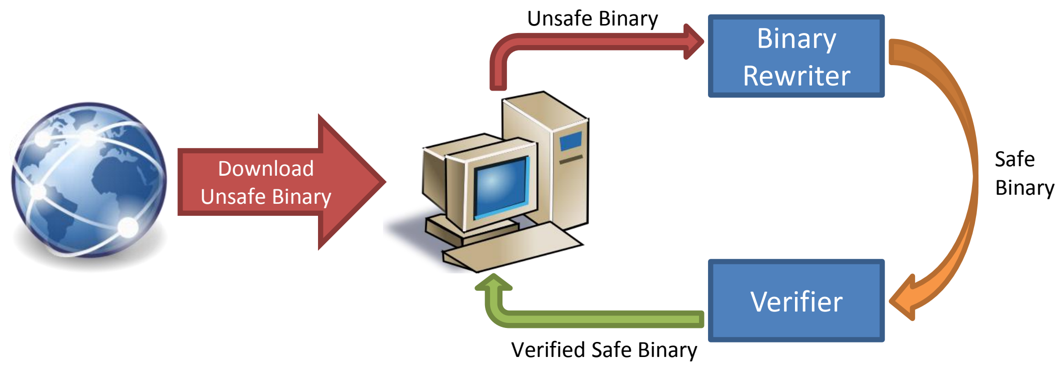


Figure 1.1. The process of rewriting an unsafe binary

5. VMs cannot be deployed as a service without expanding the circle of trust to include the service provider. For example, running software remotely on a cloud requires adding the cloud to the trusted computing base. Most clouds don't divulge details of the internal cloud architecture, so this trust is blind.

Binary transformation is an alternative solution, solving the problem of untrusted code instrumenting an untrusted binary to enforce security policies. Figure 1.1 displays the general approach of rewriting an untrusted binary from the internet.

Binary transformation comes with some strong benefits. For example, unlike a VM, a binary rewriter can enforce fine-grained security policies, since the enforcement mechanism resides within the binary instead of outside of it. In addition, different binary rewriters can enforce different policies, such as having an email client-specific rewriter that enforces different policies than a web browser-specific rewriter, allowing rewritten processes to run natively and therefore communicate directly (if permitted). This separation is not possible with a VM since it requires inter-operating processes to cohabitate under the umbrella of a single VM. Rewritten binaries are amenable to formal, machine-verification of safety, allowing the rewriter to remain untrusted, minimizing the trusted computing base. Since it's untrusted, the rewriter can be deployed as an untrusted service.

Though binary rewriting extends many benefits, it also raises daunting challenges. The first step in any binary rewriting process is accurate static analysis, including disassembly. This step is necessary to provide the instruction semantics to the rewriter, in order for it to make proper decisions about which instructions or behaviors to guard. However, x86 static disassembly is provably undecidable in the general case due to unaligned instructions, code and data interleaving, and computed control-flows. Past works overcome this challenge by restricting themselves to toy programs not representative of real-world COTS applications, requiring insider information (e.g., debug symbols) that most code producers are not willing to provide, or forcing the code producer to recompile their binary with a special compiler, which most code producers are not willing to do.

My Thesis. This dissertation argues that it is possible to perform effective, provably safe rewriting of a large category of COTS x86 binaries without metadata or perfect disassembly. Disassembly flaws can be tolerated via use of a conservative disassembler and by implementing dynamic control-flow patching that in-lines a light-weight logic to patch undiscovered computed control-flows at runtime. This approach avoids the undecidability of x86 disassembly by deferring computed control-flow decisions to runtime and including all valid execution paths in the rewritten binary. The rest of this dissertation describes the steps necessary to accomplish this through three novel approaches to disassembly and two secure rewriting strategies—one based on basic block randomization and the other using machine verifiable software fault isolation. Both of these binary rewriting techniques have very low overhead and require no source code or metadata. Lastly, it shows how compatibility issues can be handled via transparency verification.

The rest of this dissertation is structured as follows. Chapter 2 details the x86 instruction set and the difficulties involved in performing static disassembly, as well as two possible solutions. The challenges faced when performing static x86 binary rewriting are discussed in Chapter 3, as well as two interesting binary rewriters, Stir and Reins, developed for this dissertation. Behavioral equivalence of binary rewriters is discussed in Chapter 4, and a solution to transparency for ActionScript binaries is presented. Finally, relevant related work is presented in Chapter 5 and conclusions are presented in Chapter 6.

CHAPTER 2

STATIC X86 DISASSEMBLY

Disassemblers transform machine code into human-readable assembly code. For x86 executables, this can be a daunting task in practice. Unlike Java bytecode and RISC binary formats, which separate code and data into separate sections or use fixed-length instruction encodings, x86 permits interleaving of code and static data within a section and uses variable-length, unaligned instruction encodings. This trades simplicity for brevity and speed, since more common instructions can be assigned shorter encodings by architecture designers.

Since instructions are unaligned, any address can be treated as the beginning of an instruction, and thus multiple execution paths through a binary exist. Table 2.1 shows an example of *instruction aliasing*, how x86 instructions can be interleaved within each other. Malicious code is therefore much easier to conceal in x86 binaries than in other formats. To detect and identify potential attacks or vulnerabilities in software programs, it is important to have a comprehensive disassembly for analyzing and debugging the executable code.

In software development contexts, robust disassembly is generally achieved by appealing to binary debugging information (e.g. symbol tables, relocation information, etc.) that is

Table 2.1. x86 Instruction Aliasing

`0F`	`8A`	`C3`	`C0`	`E8`	`03`
`jp C3C0E803h`					
	`mov al, bl`				
		`retn`			
			`shr al, 3`		
				`call 03...`	
					`add ...`

generated by most compilers during the compilation process. However, such information is typically withheld from consumers of proprietary software in order to discourage reverse engineering and to protect intellectual property. Thus, debugging information is not available for the vast majority of COTS binaries and other untrusted mobile code to which reverse engineering is typically applied.

Without any debugging information, accurate static disassembly of an x86 binary is an undecidable problem, since reachability of arbitrary byte addresses is equivalent to the halting problem. However, in order to correctly rewrite any binary, the first step is accurate disassembly. If any of the semantic information about the code section has been lost, that information will not be translated to the rewritten file or instruction guards and optimizations may not be properly applied, causing crashes in the rewritten binary.

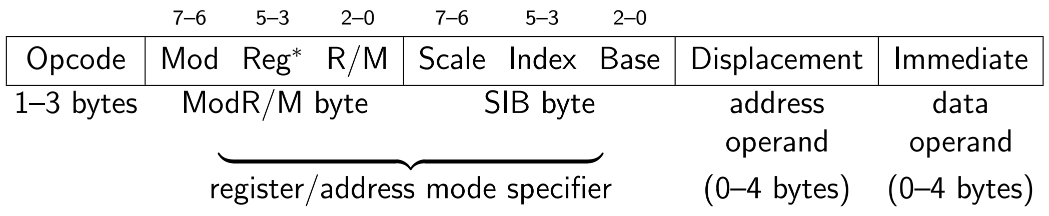
The rest of Chapter 2 is arranged as follows. Section 2.1 details the various challenges involved in performing static x86 disassembly. Current approaches to x86 disassembly are presented in Section 2.2. Finally, three different approaches to x86 disassembly that this dissertation contributes are described in Sections 2.3, 2.4, and 2.5.

Acknowledgement must be made to Dr. Yan Zhou for her work on the writing and implementation of the disassemblers in Sections 2.4 and 2.5. Dr. Kevin Hamlen was also a key contributor of ideas and writing throughout this chapter.

2.1 Challenges

Instruction Format

Figure 2.1 shows the x86 machine instruction binary format [Intel Corporation, 2012]. Instructions begin with 1–3 *opcode* bytes that identify the instruction. Instructions with operands are then followed by an *addressing form specifier* (ModR/M) byte that identifies register or memory operands for the instruction. Some addressing forms require a second



*The Reg field is sometimes used as an opcode extension field.

Figure 2.1. The x86 machine instruction format.

scale-index-base (SIB) byte that specifies a memory addressing mode. The addressing mode essentially encodes a short formula that dynamically computes the memory operand at runtime. For example, addressing mode `[eax*4]+disp32` references a memory address obtained by multiplying the contents of the `eax` register by 4 and then adding a 32-bit *displacement* constant. The displacement, if present, comes after the SIB byte. Finally, immediate operands (constants) are encoded last and have a width of up to 4 bytes (on 32-bit architectures).

In addition to this complicated instruction format, there are a number of prefix bytes that may precede the opcode bytes, all eleven of which may be used in combination and each with their own specific function. For example, `0x66` and `0x67` modify the byte width of the operand and address size respectively, where as `0xF2` and `0xF3` are both used to repeat the instruction they precede a specific number of times as specified in the `ecx` register.

A few x86 machine instructions have multiple different correct representations at the assembly level. Most notable is the floating point `WAIT` instruction, which can either be interpreted as an opcode prefix for the instruction it precedes, or as a separate instruction in its own right. We adopt the former interpretation in our treatment, since it makes for a more compact assembly representation.

To contrast, Figure 2.2 shows the instruction format of Java bytecode. The drastic difference in complexity is apparent. Each Java bytecode instruction starts with a 1 byte opcode which is optionally followed by operand bytes depending on the opcode. There

Opcode	Operand
1 byte	(0+ bytes)

Figure 2.2. The Java bytecode instruction format.

are only 256 possible opcodes (`0x00-0xFF`), and 54 are currently unused. In addition, java bytecode branch targets are static, so instruction aliasing is not possible.

Control-Flow Modifying Instructions

Instructions that modify the control-flow of a program (that modify the `eip` register) can either be *direct* or *indirect*.

Definition 2.1.1. ***Direct Branches***
Direct branch *instructions add their operand to the* `eip` *register, treating it as an offset.*

Definition 2.1.2. ***Indirect Branches***
Indirect branch *instructions calculate their targets at runtime, either by setting the* `eip` *register to their operand, as in the case of* `jmp` *and* `call` *instructions, or popping their target off the stack in the case of* `retn` *instructions.*

Direct branch instructions are not difficult to analyze as their target is static and does not change at runtime. Indirect branches, however, calculate their targets at runtime. Register values, stack values, and any dereferenced memory address can be targeted. Table 2.2 shows examples of each indirect branch type. Due to these branch types and the lack of restrictions on branch targets, all addresses within an executable code section are viable targets, even if they are in the interior of another instruction.

Once again, we contrast this property against Java bytecode, which disallows branch instructions to the middle of another instruction. In addition, branches are all direct in Java bytecode, and thus can all be determined statically.

Table 2.2. Indirect Branch Types

Branch Type	**Example**
Register	`call eax`
Dereferenced Register	`call [eax]`
Dereferenced Stack	`call [esp - 4]`
Dereferenced Memory	`call [0x4040d4]`
Scaled Dereferenced Memory	`call [eax*4 + 0x4040d4]`

Code/Data Interleaving

The final property that makes x86 binaries so difficult to disassemble is that code and data can be interleaved within executable sections. Consider Java bytecode, which disallows such interleaving: disassembly starts with the first instruction and follows to each successive instruction until the end is reached. The same would be true in x86 if there was no data within executable sections. Every byte in the executable section can be treated as data or as code, and theoretically as both.

Table 2.3 shows the differences between disassembling java and x86 binaries. Multiple execution sequences exist for the same sequence of bytes in x86, where as in Java bytecode, only one instruction sequence is possible.

Static x86 Disassembly Undecideability

The combination of these factors implies the following lemma about static x86 disassemblies:

Lemma 2.1.3. *It is Turing-undecidable whether an arbitrary assembly code is an accurate disassembly of an arbitrary, executable x86 binary, where an accurate disassembly is one that correctly classifies each byte as (a) an instruction boundary, (b) internal to an instruction, or (c) non-executable data.*

Byte Sequence

`5E 5F 5D C3 E8 BB 1C 03 00 89 04 24 68 BB 1C 03`

Java Disassembly	**x86 Disassembly 1**	**x86 Disassembly 2**	**x86 Disassembly 3**
`dup2_x2`	`pop esi`	`pop esi`	`pop esi`
`new`	`pop edi`	`pop edi`	`pop edi`
`nop`	`pop ebp`	`pop ebp`	`pop ebp`
`l2f`	`retn`	`retn`	`retn`
`iconst_1`	`call sub_433784`	`db EB`	`db EB`
`fload_2`	`mov [esp+24], eax`	`mov ebx, 8900031Ch`	`db BB`
`imul`	`push BB1C03A1`	`add al, 24h`	`sbb al, 3`
`new`		`push BB1C03A1`	`add [ecx-44...], cl`
			`sbb al, 3`

Figure 2.3. Java Disassembly vs. x86 Disassembly

2.2 Current x86 Disassembly

There are a variety of different approaches to statically disassembling an x86 binary accurately. Table 2.3 gives examples of the most common approaches, as well as some of their benefits and deficiencies.

Modern static disassemblers for x86 binaries employ a variety of techniques to accurately differentiate bytes that comprise instructions from those that comprise static data. IDA Pro [Hex-Rays, 2012] is widely acknowledged as the best tool currently available for distinguishing code from data in arbitrary binaries (cf., [Balakrishnan et al., 2005, Kinder and Veith, 2008]). It combines straight-line, heuristic, and execution emulation-based disassembly while also providing an extensive GUI interface and multiple powerful APIs for interacting with the disassembly data. Recent work has applied model-checking and abstract interpretation to improve upon IDA Pro's analysis [Kinder and Veith, 2008, Kinder et al., 2009], but application of these technologies is currently limited to relatively small binaries, such as device drivers, for which these aggressive analyses remain tractable.

Table 2.3. Disassembly Techniques

Disassembly technique	**Description**
Fall-through	Disassemble from the first byte of the binary and fall through to each following instruction, regardless of semantics. No attempt to distinguish code from data is made.
Control-flow	Follow the semantics of the assembly language, using branches to determine where to disassemble next. However, computed jump targets are difficult to determine because they are calculated at run-time.
Heuristics	Probability based heuristics can be used in order to disassemble x86, based on the semantics of x86. Since these are based on probabilities, many false positives arise in distinguishing code from data.
Dynamic	Disassembling dynamically is a viable solution for many problems, but a code-coverage issue arises due to the fact that most executions do not actually execute all instructions in the code section of a binary.

All other widely available disassemblers to our knowledge take a comparatively simplistic approach that relies mainly upon straight-line disassembly, and that therefore requires the user to manually separate code from data during binary analysis. Our tests therefore focus on comparing the accuracy of our algorithm to that of IDA Pro.

Disassembly heuristics employed by IDA Pro include the following:

- *Code entry point.* The starting point for analyzing an executable is the address listed in the header as the code entry point. That address must hold an instruction, and will hopefully lead to successfully analyzing a large portion of the executable.

- *Function prologues and epilogues.* Many function bodies compiled by mainstream compilers begin with a recognizable sequence of instructions that implement one of the standard x86 calling conventions. These byte sequences are assumed by IDA Pro to be the beginnings and ends of reachable code blocks.

- *Direct jumps and calls.* The destination address operand of any static jump instruction that has already been classified as reachable code is also classified as reachable code.

- *Unconditional jumps and returns.* Bytes immediately following a reachable, unconditional jump or return instruction are considered as potential data bytes. These often contain static data such as jump tables, padding bytes, or strings.

However, despite a decade of development and tuning, IDA Pro nevertheless fails to reliably distinguish code from data even in many non-malicious, non-obfuscated x86 binaries. Some common mistakes include the following:

- *Misclassifying data as returns.* IDA Pro frequently misclassifies isolated data bytes within data blocks as return instructions. Return instructions have a one-byte x86 encoding and are potential targets of computed jumps whose destinations are not statically decidable. This makes them extremely difficult to distinguish from data. IDA

Pro therefore often misidentifies data bytes that happen to match the encoding of a return instruction.

- *16-bit legacy instructions.* The x86 instruction set supports legacy 16-bit addressing modes, mainly for reasons of backward compatibility. The vast majority of genuinely reachable instructions in modern binaries are 32- or 64-bit. However, many data bytes or misaligned code bytes can be misinterpreted as 16-bit instructions, leading to flawed disassemblies.

- *Mislabeled padding bytes.* Many compilers generate padding bytes between consecutive blocks of code for alignment purposes. These bytes are not reached by typical runs, nor accessed as data, so their proper classification is ambiguous. IDA Pro typically classifies them as data, but this can complicate some code analyses by introducing many spurious code-data boundaries in the disassembly. In addition, these bytes can later become reachable if the binary undergoes hotpatching [Microsoft Corporation, 2005]. We therefore argue that these bytes are more properly classified as code.

- *Flows from code to data.* IDA Pro disassemblies frequently contain data bytes immediately preceded by non-branching or conditionally branching instructions. This is almost always an error; either the code is not actually reachable (and is therefore data misidentified as code) or the data is reachable (and is therefore code misidentified as data). The only exception to this that we have observed in practice is when a call instruction targets a non-returning procedure, such as an exception handler or the system's process-abort function. Such call instructions can be immediately followed by data.

To provide a rough estimate of the classification accuracy of IDA Pro, we wrote scripts in IDAPython [Erdélyi, 2008] that detect obvious errors made by IDA Pro in its disassemblies.

Table 2.4. Statistics of IDA Pro 5.5 disassembly errors

File Name	**Instructions**	**Mistakes**
Mfc42.dll	355906	1216
Mplayerc.exe	830407	474
RevelationClient.exe	66447	36
Vmware.exe	364421	183

Table 2.4 gives a list of the executables we tested and counts of the errors we identified for IDA Pro 5.5. The main heuristic we used to identify errors is the existence of a control-flow from code to data. Certain other errors were identified via manual inspection. It is interesting to note that most programs compiled using the Gnu family of compilers have little to no errors in their IDA Pro disassemblies. This is probably because Gnu compilers tend to yield binaries in which code and data are less interleaved, and they perform fewer aggressive binary-level optimizations that can result in code that is difficult to disassemble.

2.3 Shingled Disassembly

Since computed branch instructions in x86 have their targets established at runtime, every byte within the code section can be a target and thus must be considered as executable code. Therefore, we refer to a disassembler that retains all possible execution paths through a binary as a *Shingled Disassembler*.

Definition 2.3.1. ***Shingle***
A shingle *is a consecutive sequence of bytes that decodes to a single machine instruction. Shingles may overlap.*

The core functionality of the shingled disassembler is to eliminate bytes that are clearly data and compose a byte sequence that retains information for generating every possible valid shingle of the source binary. This is a major benefit of this approach since the shingled disassembly encodes a superset of all the possible valid disassemblies of the binary. In later sections, we discuss how we apply our graph disassembler to prune this superset until we find the most probable execution paths. In order to define what consists of a valid execution path, we must first discuss a few key concepts.

Definition 2.3.2. ***Fall-through***

Shingle x (conditionally) falls through *to shingle y, denoted $x \rightharpoondown y$, if shingle y is located adjacent to and after instruction x, and the semantics of instruction x do not (always) modify the program counter*[1]*. In this case, execution of instruction x is (sometimes) followed by execution of instruction y at runtime.*

Definition 2.3.3. ***Unconditional Branch***

A shingle is an unconditional branch *if it only falls through when its operand explicitly targets the immediately following byte. Unconditional branch instructions for x86 include `jmp` and `ret` instructions.*

Unconditional branch instructions are important in defining valid disassemblies because the last instruction in any disassembly must be an unconditional branch. If this is not the case, the program could execute past the end of its virtual address space.

Definition 2.3.4. ***Static Successor***

A control-flow edge (x, y) is static *if $x \rightharpoondown y$ holds or if x is a conditional or unconditional branch with fixed (i.e., non-computed) destination y. An instruction's* static successors *are defined by $S(x) = \{y \mid (x, y) \text{ is static}\}$.*

Definition 2.3.5. ***Post-dominating Set***

The (static) post-dominating set $P(x)$ *of shingle x is the transitive closure of S on $\{x\}$. If*

[1]At first glance, it would seem that we could strengthen our definition of fall-throughs to any two instructions that do not have an unconditional branch instruction between them. However, there are cases where a compiler will place a `call` and `jcc` instruction followed by data bytes. A common example of this is `call [IAT:ExceptionHandler]` since the exception handler function will never return.

there exists a static control-flow from x to an illegal address (e.g., an address outside the address space or whose bytes do not encode a legal instruction), then $P(x)$ is not well defined and we write $P(x) = \perp$.

Definition 2.3.6. ***Valid Execution Path***

All paths in $P(x)$ are valid execution paths *from x.*

The x86 instruction set does not make use of every possible opcode sequence, therefore certain bytes cannot be the beginning of a code instruction. For example, the `0xFF` byte is used to distinguish the beginning of one 7 different instructions, using the byte that follows to distinguish which instruction is intended. However, `0xFFFF` is an invalid opcode that is unused in the instruction set. This sequence of bytes is common because any negative offset in two's complement that branches less than `0xFFFF` bytes away will start with `0xFFFF`. The shingled disassembler can immediately mark any shingle whose opcode is not supported under the x86 instruction set as data. A shingle that is marked as data is either used as the operand of another instruction, or it is part of a data block within the code section. Execution of the instruction would cause the program to crash.

Lemma 2.3.7. ***Invalid Fall-through***

$\forall x, y :: x \rightharpoondown y \land y = \emptyset \rightarrow x = \emptyset$*, in which $\emptyset$ stands for data bytes.*

Any time that we encounter an address that is marked data, all fall-throughs to that instruction can be marked as data as well. Direct branches also fall into this definition. All direct `call` and `jmp` instructions imply a direct executional relationship between the instruction and its target. Therefore, if any shingle that targets a shingle previously marked as data can also be marked as data.

Definition 2.3.8. ***Sheering***

A shingle x is sheered *from the shingled disassembly when $\forall y :: x \rightharpoondown y$, x and all y are marked as data in the shingled disassembly.*

Figure 2.4 illustrates how our shingled disassembler works. Given a binary of byte sequence `6A01 5156 8BC7 E8B6 E6FF FF...`, the shingled disassembler performs a single-pass, ordered scan over the byte sequence. Data bytes and invalid shingles are marked along the way. Figure 2.4a demonstrates the first series of valid shingles, beginning at the first byte of the binary. Figure 2.4b starts at the second byte, which falls through to a previously disassembled shingle. The shingle with byte `C7` is then marked as data (shaded in Figure 2.4) since it is an invalid opcode. Figure 2.4c shows an invalid shingle since it falls through to an invalid opcode `FFFF`. Our shingled disassembler marks the two shingles `B6` and `FF` as invalid in the sequence. Figure 2.4d shows another valid shingle that begins at the ninth byte of the binary. After completing the scan, our shingled disassembler has stored information necessary to produce all valid paths in $P(x)$.

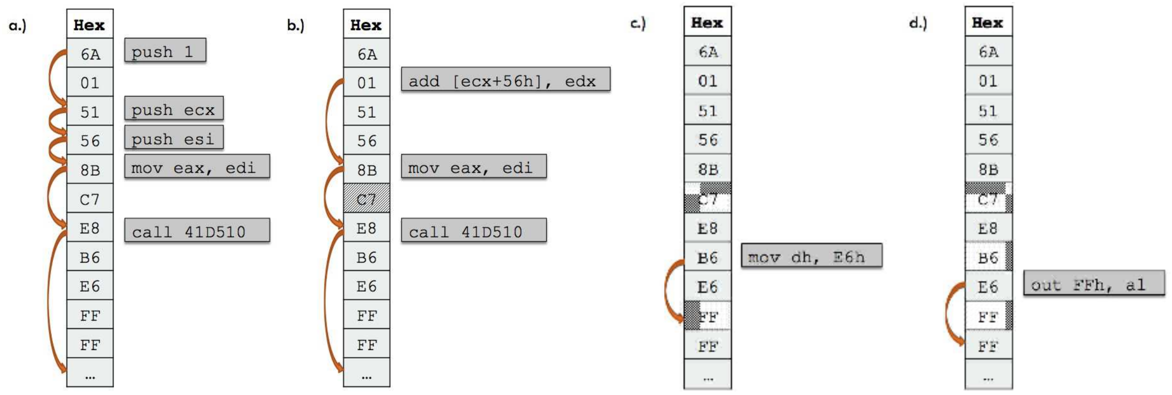


Figure 2.4. Shingled disassemblies of a given binary `6A0151568BC7E8B6E6FFFF` (a) a sequence of shingles beginning at the first byte; (b) a sequence of shingles beginning at the second byte; (c) an invalid shingle that falls through to an invalid opcode `FF`; (d) a valid shingle beginning at the `E6` byte.

The pseudocode for generating a shingled disassembly for a binary is shown in Algorithm 1.

This disassembly technique results in a completely valid disassembly that executes properly. All instruction sequences included in the shingled disassembly are valid based on the semantics of the x86 instruction set. However, not all of the disassembled shingles are necessary: many of the instructions in the disassembly are never executed and were not intended execution

Algorithm 1: Shingled Disassembly

```
Input: $x_0 \ldots x_i \ldots x_{n-1}$ ;                                   // byte input
       $Ins(x_i)$—output $\emptyset$ if $x_i$ is a data byte
Output: $y_0 \ldots y_i \ldots y_{n-1}$
$t = 0$ ;
$y_i = \varnothing, \forall i = 1, \ldots, n-1$ ;                          // initialize $y_i$
while $t < n$ do
    $v = t$ ;
    while $y_v = \varnothing$ do
        $y_v = Ins(x_v) \parallel \emptyset$;
        if $y_v \neq \emptyset$ then
            $v = v + \mid Ins(x_v) \mid$ ;
        end
        else
            break ;
        end
    end
    $t++$;
end
$t = n$ ;
while $t \geq 0$ do
    if $x_t = \emptyset$ then
        // invalidate any opcode branching to address t
        $\forall v$ if $v + \mid x_v \mid := t$ then
            $x_v := \emptyset$ ;
        end
    end
    $t--$;
end
```

sequences when the binary was compiled. In order to prune unnecessary shingles, another disassembly technique can be applied to narrow down to a single execution sequence of shingles.

2.4 Machine Learning Disassembly Model

None of the approaches listed in Section 2.2 attempt to use Machine Learning as a tool in the disassembly process. Each of the described techniques is based around a knowledge of the semantic information in an executable's code section (or ignorance of it in the case of straight-line disassembly). However, the problem of disassembly can be simplified down beyond instruction semantics.

We define the *tagging problem* as follows: Given a non-empty input string X over an alphabet Σ, find a set of transition events $\mathcal{T} = \{\$_1, \ldots, \$_M\}$ such that $\mathcal{T} = \arg\max_{\mathcal{T}} f(X, \mathcal{T})$, where $\$_i$ at position $i < |X|$ marks a transition event e in X, and f is a function that measures the likelihood that X is tagged correctly.

The tagging problem resembles the word segmentation problem in some natural languages where no clear separations exist between different words [Teahan et al., 2000]. In the word segmentation problem, the task is to find correct separations between sequences of characters to form words. In the tagging problem, our objective is to find separations between different instructions, and often between instructions and data as well. In both problems, resolving ambiguities is the major challenge. For example, a byte sequence `E8 F9 33 6A 00` can be a 5-byte call instruction (opcode `E8`), or three bytes of data followed by a push instruction (opcode `6A`). Ambiguities can only be resolved through investigating their surrounding context.

Solutions to the tagging problem must also successfully identify and ignore "noise" in the form of padding bytes. Padding bytes are neither executed as code nor accessed as data on any run of the executable, so their classification is ambiguous. However, reliably distinguishing these padding sequences from true code and data is non-trivial because the same sequence of bytes often appears as both code and padding within the same executable. For example, the instruction

```
8D A4 24 00 00 00 00    lea esp, [esp+0x0]
```

is semantically a *no-operation* (NOP), and is therefore used as padding within some instruction streams to align subsequent bytes to a cache line boundary, but is used in other instruction streams as a genuinely reachable instruction. Another common use of semantic NOPs is to introduce obfuscation to hide what the program is doing.

In general, code and data bytes may differ only in their locations in the sequence, not in their values. Any byte sequence that is code could appear as data in an executable, even though it should statistically appear much more often as code than data. Not every data sequence can be code, however, since not all byte sequences are legitimate instruction encodings, as was discussed in Section 2.3.

2.4.1 Design

There are three components in our tagging algorithm: an instruction reference array, a utility function, and heuristic sanity checks. The reference array stores the length of an instruction given the bytes of an opcode (and the existence of length-relevant prefix bytes). The utility function estimates the probability that a byte sequence is code. We estimate the probability using a context-based language model built from pre-tagged x86 executables. Finally, our sanity checks use heuristics to verify that our disassembly does not violate x86 instruction semantics.

Instruction Reference Array

From the x86 instruction decoding specification we derive a mapping from the bytes of an opcode to the length of the instruction. This is helpful in two respects: First, it marks a definite ending of an instruction that allows us to move directly to the next instruction or data. Second, it tells us when a series of bytes is undefined in the x86 instruction set, which means that the current byte cannot be the beginning of an instruction. We tested our code against more than ten million instructions in the IDA Pro disassembler and had 100% accurate instruction lengths.

Utility Function

The utility function helps predict whether a byte sequence is code or data in the current context. If the current byte sequence is unlikely to be code, our tagging algorithm moves to the next byte sequence. If we predict that the byte sequence is code, we look up the length of the instruction in the instruction reference array and move to the next byte sequence. The following two properties express the desired relationship between the utility function and its input byte sequence.

Property 2.4.1. *A byte sequence bordered by transitions is tagged as code (resp., data) if its utility as code (resp., data) is greater than its utility as data (resp., code).*

Property 2.4.2. *A transition between two byte sequences S_A and S_B entails a semantic ordering in machine code: $f(S_B|S_A) \geq f(S_B|S_*)$, where S_* is any subsequence but S_A in a given binary, and f is the utility function.*

Our utility function estimates the likelihood of a transition event using context-based analysis. We collect context statistics from a set of pre-tagged binaries in the training set. In a pre-tagged binary, code-code and code-data/data-code transitions are given. Two important forms of information are yielded by pre-tagged binaries. First, they provide semantic groupings of byte sequences that are either code or data; and second, they provide a semantic ordering between two subsequences, which predicts how likely a subsequence is followed by another. To correctly tag an input hex string, both pieces of information are important. This calls for a language model that We choose to use a context-based statistical data compression model for storing context statistics of each byte sequence encountered in the training set of binary executables. base of the utility function. The data model have the following traits:

- can capture local coherence in a byte sequence, and
- can capture long-range correlations between two adjacent subsequences—i.e., subsequences separated by a code-code or code-data/data-code transition.

Several modern statistical data compression models [Moffat and Turpin, 2002] are known for their context-based analysis. These data models can work directly on any raw input regardless of source and type. We use the current state of the art data compression model as our language model. Before we discuss the details of the language model, we give the tagging algorithm in Algorithm 2.

Algorithm 2: Tagging

```
Input: $x_0 \dots x_i \dots x_{n-1}$ ;                                  // input string of bytes
       $M_c$ ;                                                              // language model
Output: $x_0 \dots x_i | x_{i+1} \dots x_j | \cdots | x_k \dots x_{n-1}$ ;     // segmented string
$t \leftarrow 0$ ;
while $t < n$ do
    $\ell \leftarrow 0$;
    if $x_t \in M_c$ then
        $\ell \leftarrow codeLength(x_t \dots x_{\min\{t+4,n-1\}})$ ;     // lookup instruction length
    if $(\ell = 0) \vee (t + \ell > n)$ then $\ell \leftarrow 1$ ;          // tag as possible data
    print $x_t \dots x_{t+\ell-1}$ ;                                     // output the segment
    $t \leftarrow t + \ell$;
```

Context-based Data Compression Model. The compression model we use to store context statistics is *predication by partial matching* (PPM) [Cleary and Witten, 1984, Cormack and Horspool, 1987, Cleary and Teahan, 1997]. The theoretical foundation of the PPM algorithm is the kth order Markov model, where k constrains the maximum order context based on which a symbol probability is predicted. PPM models both short-range and long-range correlations among subsequences by using dynamic context match. The context of the ith symbol x_i in an input string is the previous $i-1$ symbols. Its kth order context c_i^k includes only the k prior symbols. To predict the probability of seeing x_i in the current location of the input, the PPM algorithm first searches for a match of c_i^k in the context tree. If a match is found, $p(x_i|c_i^k)$ is returned as the symbol probability. If such a match does not exist in the context tree, an *escape event* is recorded and the model falls back to a lower-order context c_i^{k-1}. If a match is found, the following symbol probability is returned:

$$p(x_i|c_i^k) = p(Esc|c_i^k) \cdot p(x_i|c_i^{k-1})$$

where $p(Esc|c_i^k)$ is the escape probability conditioned on context c_i^k. The *escape probability* models the probability that x_i will be found in the lower-order context. This process is repeated whenever a match is not found until an order-0 context has been reached. If x_i appears in the input string for the first time, a uniform probability of distinct symbols that have been observed so far will be returned. Therefore, the probability of x_i in a string of input is modeled as follows:

$$p(x_i|c_i^k) = \begin{cases} \left(\prod_{j=k'+1}^{k} p(Esc|c_i^j)\right) \cdot p(x_i|c_i^{k'}) & \text{if } k \geq 0 \\ \frac{1}{|A|} & \text{if } k = -1 \end{cases}$$

where $k' \leq k$ is the context order when the first match is found for x_i, and $|A|$ is the number of distinct symbols seen so far in the input. If the symbol is not predicted by the order-0 model, a probability defined for the order -1 context is predicted.

The PPM model predicts symbol probabilities. To estimate the probability of a sequence of symbols, we compute the product of the symbol probabilities in the sequence. Thus, given a data sequence $X = x_1x_2\ldots x_d$ of length d, where x_i is a symbol in the alphabet, the probability of seeing the entire sequence given a compression model M can be estimated as

$$p(X|M) = \prod_{i=1}^{d} p(x_i|x_{i-k}^{i-1})$$

where $x_i^j = x_ix_{i+1}x_{i+2}\ldots x_j$ for $i < j$.

We use the above probability estimate as our utility function. We build two compression models M_c and M_d from the pre-tagged binaries in the training set: M_c is built from tagged instructions and M_d is built from tagged data. Given a new binary executable e and a subsequence e_i in e,

$$M_c = \{e_i \,|\, p(e_i|M_c) > p(e_i|M_d)\}$$

Classification. After tagging the transitions in the executable, we have segments of bytes. Even though the tagging algorithm outputs each segment either as code or data, we cannot assume this preliminary classification is correct because some data bytes may match legitimate opcodes for which a valid instruction length exists in the reference array. The tagging algorithm will output this segment as code even though it is data. Therefore, we need to reclassify each segment as data or code.

Our classification algorithm makes use of the aforementioned language model and several well known semantic heuristics. The language models are also used in the tagging algorithm. The heuristics are adapted from those used by human experts for debugging disassembly errors. We first discuss the language model-based classification module followed by the semantic heuristics.

Classification Using Language Model. Classifying byte sequences is a binary classification problem. We reuse the two compression models built for tagging. Recall that model M_c is built from pre-tagged code and model M_d is built from the pre-tagged data in the training set. To classify a byte sequence B, we compute a log likelihood of B using each data model $\alpha \in \{c, d\}$:

$$p(B|M_\alpha) = -\log \prod_{i=1}^{|B|} p(b_i | b_{i-k}^{i-1}, M_\alpha)$$

where M_α is the compression model associated with class α, $|B|$ is the length of byte sequence B, sequence $b_{i-k}, \ldots, b_i$ is a subsequence in B, and k is the length of the context. The class membership α of B is predicted by minimizing the cross entropy [Teahan, 2000, Bratko et al., 2006]:

$$\alpha = \arg\min_{\alpha \in \{c,d\}} -\frac{1}{|B|} p(B|M_\alpha)$$

Sanity Checks

Once we have deemed which instructions are code and data, sanity checks are necessary to ensure that the disassembly we have generated makes sense based on x86 instruction semantics.

Certain semantic heuristics are helpful in determining an accurate class membership of an x86 byte sequence. Reverse engineers rely heavily upon such heuristics when manually correcting flawed disassemblies. In most cases, these heuristics make no changes but are useful in ensuring the disassembly is creating a valid execution sequence.

Word data tables. Many static data blocks in code sections store tables of 4-byte integers. Often the majority of 4-byte integers in these tables have similar values, such as when the table is a method dispatch or jump table consisting of code addresses that mostly lie within a limited virtual address range. One way to quickly identify such tables is to examine the distribution of byte values at addresses that are 1 less than a multiple of 4. When these high-order bytes have low variance, the section is likely to be a data table rather than code, and is classified accordingly.

16-bit addressing modes. When classifying a byte sequence as code yields a disassembly densely populated by instructions with 16-bit operands (and the binary is a 32-bit executable), this indicates that the sequence may actually be data misclassified as code. Modern x86 architectures support the full 16-bit instruction set of earlier processor generations for backward compatibility reasons, but these legacy instructions appear only occasionally in most modern 32-bit applications. The 16-bit instructions often have short binary encodings, causing them to appear with higher frequency in randomly generated byte sequences than they do in actual code.

Data after unconditional jumps. Control-flows from code to data are almost always disassembly errors; either the data is reachable and is therefore code, or the code is actually unreachable and is therefore data. Thus, data inside of a code section can only occur at the very beginning of the section or after a branch instruction—usually an unconditional jump or return instruction. It can occasionally also appear after a call instruction if the call

Table 2.5. Programs tested with PPM Disassembler

File Name	**File Size (K)**	**Code (K)**	**Data (K)**	**Transitions**
`7zFM.exe`	379	271	3.3	1379
`notepad.exe`	68	23	8.6	182
`DosBox.exe`	3640	2947	67.2	15355
`WinRAR.exe`	1059	718	31.6	5171
`Mulberry.exe`	9276	4632	148.2	36435
`scummvm.exe`	11823	9798	49.2	47757
`emule.exe`	5624	3145	119.5	24297
`Mfc42.dll`	1110	751	265.5	15706
`Mplayerc.exe`	5858	4044	126.1	28760
`RevelationClient.exe`	382	252	18.4	1493
`Vmware.exe`	2675	1158	87.3	18259

never returns (e.g., the call targets an exception handler or process-abort function). This observation gives rise to the following heuristics:

- If an instruction is a non-jump, non-return surrounded by data, it is reclassified as data.
- If a byte sequence classified as data encodes an instruction known to be a semantic NOP, it is reclassified as code.

2.4.2 Evaluation

We tested our disassembly algorithm on the 11 real-world programs listed in Table 2.5. In each experiment, we used 10 of the programs to build the language models and the remaining one for testing. All the executables are pre-tagged using IDA Pro; however, IDA Pro yields imperfect disassemblies for all 11 executables. Some instructions it consistently labels as data, while others—particularly those that are semantic `nops`—it labels as data or code depending on the context. This leads to a noisy training set.

Since we lack perfect disassemblies of any of these programs, evaluation of the classification accuracy of each algorithm is necessarily based on a manual comparison of the disassembly results. When the number of classification disagreements is large, this can quickly exceed the human processing limit. However, disagreements in which one algorithm identifies a large, contiguous code section missed by the other are relatively easy to verify by manual inspection. These constituted the majority of the disagreements, keeping the evaluation tractable.

Tagging Results. We first report the accuracy of our tagging algorithm. Inaccuracies can take the form of code misclassified as data (false negatives) and data misclassified as code (false positives). Both can have potentially severe consequences in the context of reverse engineering for malware defense. False negatives withhold potentially malicious code sequences from expert analysis, allowing attacks to succeed; false positives increase the volume of code that experts must examine, exacerbating the difficulty of separating potentially dangerous code from benign code. We therefore compute the tagging accuracy as

$$accuracy = 1 - \frac{false\ negatives + false\ positives}{total\ number\ of\ instructions}$$

where false positives count the number of instructions erroneously disassembled from data bytes.

As can be seen in Table 2.6 we were able to tag 6 of the 11 binaries with 100% accuracy. For the remaining 5, the tagging errors were mainly caused by misclassification of small word data tables (see §2.4.1) consisting of 12 or fewer bytes. Our heuristic for detecting such tables avoids matching such small tables in order to avoid misclassifying short semantic NOP sequences that frequently pad instruction sequences. Such padding often consists of 3 identical 4-byte instructions, which collectively resemble a very short word data table.

Classification Results. To evaluate the classification accuracy we took the output of our tagging algorithm and ran each segment through the language model to get its class membership. Table 2.7 shows the classification results of our disassembly algorithm. False positives (FP), false negatives (FN), and overall classification accuracy is listed for each disassembler.

Table 2.6. Tagging accuracy

File Name	**Errors**	**Total**	**Tagging Accuracy**
`7zFM.exe`	0	88164	100%
`notepad.exe`	0	6984	100%
`DosBox.exe`	0	768768	100%
`WinRAR.exe`	39	215832	99.982%
`Mulberry.exe`	0	1437950	100%
`scummvm.exe`	0	2669967	100%
`emule.exe`	117	993159	99.988%
`Mfc42.dll`	0	355906	100%
`Mplayerc.exe`	307	830407	99.963%
`RevelationClient.exe`	71	66447	99.893%
`Vmware.exe`	16	364421	99.998%

Table 2.7. A comparison of mistakes made by IDA Pro and by our disassembler

	IDA Pro 5.5			**Ours**		
File Name	**FP**	**FN**	**Accuracy**	**FP**	**FN**	**Accuracy**
`7zFM.exe`	0	1	99.999%	0	0	100%
`notepad.exe`	4	0	99.943%	0	0	100%
`DosBox.exe`	0	26	99.997%	0	0	100%
`WinRAR.exe`	0	23	99.989%	0	39	99.982%
`Mulberry.exe`	0	202	99.986%	0	0	100%
`scummvm.exe`	0	65	99.998%	0	0	100%
`emule.exe`	0	681	99.931%	0	117	99.988%
`Mfc42.dll`	0	1216	99.658%	0	47	99.987%
`Mplayerc.exe`	0	2517	99.697%	0	307	99.963%
`RevelationClient.exe`	0	2301	96.537%	0	71	99.893%
`Vmware.exe`	0	183	99.950%	0	45	99.988%

eMule Case Study. To show some of the specific differences between decisions made by IDA Pro's disassembler and our approach, we here present a detailed case study of eMule, a popular peer-to-peer file sharing program. Case studies for other executables in our test suite are similar to that presented here. Table 2.8 illustrates examples in which IDA Pro classified bytes were code but our disassembler determined that they were data, or vice versa. In the table, *db* is an assembly directive commonly used to mark data bytes in a code listing. To identify all discrepancies, we stored all instructions from both disassemblies to text files with code/data distinguishers before every instruction. We then used `sdiff` to find the differences. The cases in Table 2.8 summarize all of the different kinds of discrepancies we discovered.

IDA Pro makes heavy use of heuristic control-flow analysis to infer instruction start points in a sea of unclassified bytes. Thus, its classification of bytes immediately following a call instruction depends on its estimate of whether the called method could return. For example, Case 1 of Table 2.8 shows a non-returning call to an exception handler. The call is immediately followed by padding bytes that serve to align the body of the next function. These bytes are also legitimate (but unreachable) instructions, so could be classified as data or code (though we argue in §2.2 that a code classification is preferable). However, this control-flow analysis strategy leads to a classification error in Case 2 of the table, wherein IDA Pro incorrectly identifies method `GetDLGItem` as non-returning and therefore fails to disassemble the bytes that follow the call. Our disassembler correctly identifies both byte sequences as code. Such scenarios account for about 20 of IDA Pro's disassembly errors for eMule.

Case 3 of Table 2.8 illustrates a repetitive instruction sequence that is difficult to distinguish from a table of static data. IDA Pro therefore misidentifies some of the bytes in this sequence as data, whereas our algorithm correctly identifies all as code based on the surrounding context.

Many instruction sequences in x86 binaries are only reachable at runtime via dynamically computed jumps. These sequences are difficult to identify by control-flow analysis alone

Table 2.8. Disassembly discrepancies between IDA Pro and our disassembler for eMule

		Example Disassemblies	
Case	**Description**	**IDA Pro 5.5**	**Ours**

1	padding after a non-returning call	`call ExceptionHandler`
		db (1–9 bytes)
		`function_start`
			`call ExceptionHandler`
			code (1–9 bytes)
			`function_start`

2	calls misidentified as non-returning	`call GetDLGItem`
		db 88h 50h
		`sbb al, 8Bh`
			`call GetDLGItem`
			`mov edx, [eax+1Ch]`

3	repetitive instruction sequences	*db (4 bytes)*
		
		`push 0`
		`push 0`
		`call 429dd0h`
			`push 0`
			`push 0`
			`push 0`
			`push 0`
			`call 429dd0h`

4	missed computed jump targets	*db (12 bytes)*
		
		`push offset 41CC30h`
			`mov eax, large fs:0`
			`mov edx, [esp+8]`
			`push FFFFFFFFh`
			`push offset 41CC30h`

5	false computed jump targets	`push ecx`
		db FFh
		`adc eax, 7DFAB4h`
		`mov ebp, eax`
		db 8Bh
		`sbb esp, 0`
		db (13 bytes)
		
		`test esi, esi`
			`push ecx`
			`call 7DFAB4h`
			
			`mov ebp, eax`
			`mov eax, [ebx+0DCh]`
			`mov ecx, [eax+4]`
			`cmp ecx, esi`
			`jle loc_524D61`
			`test esi, esi`

6	missed opcode prefixes	`push offset 701268h`
		db 64h
		`mov eax, large ds:0`
			`push offset 701268h`
			`mov eax, large fs:0`

7	code following unconditional branches	`jmp 526396h`
		db 8Bh
		`or eax, 9CAF08h`
			`jmp 526396h`
			`mov ecx, 9CAF08h`

8	code following returns	`retn`
		db C4h 83h
		`sub al, CDh`
		`push es`
			`retn`
			`add esp, 2Ch`
			`int 6`

9	code following conditional branches	`jz 52518Fh`
		db 8Bh
		`or eax, 9CAF04h`
			`jz 52518F`
			`mov ecx, 9CAF04h`

since the destinations of dynamic jumps cannot be statically predicted in general. Case 4 is an example where IDA Pro fails to identify a computed jump target and therefore fails to classify the bytes at that address as code; however, our disassembler finds and correctly disassembles the instructions.

Misidentifying non-jump targets as possible targets leads to a different form of disassembly error. Case 5 illustrates an example in which an early phase of IDA Pro's analysis incorrectly identifies the interior byte of an instruction as a possible computed jump destination (probably because some bytes in a data section happened to encode that address). The bytes at that address disassemble to an `adc` instruction that turns out to be misaligned with respect to the surrounding sequence. This leads to an inconsistent mix of code and data that IDA Pro cannot reconcile because it cannot determine which interpretation of the bytes is correct. In contrast, our algorithm infers the correct instruction sequence, given in the rightmost column of the table.

Some instructions include prefix bytes, as discussed in §2.1. The suffix without the prefix bytes is itself a valid instruction encoding. IDA Pro's analysis sometimes misses these prefix bytes because it discovers the suffix encoding first and treats it as a self-contained instruction. This leads to the disassembly error depicted in Case 6 of the table. Our approach avoids this kind of error in all cases.

Cases 7–8 of the table illustrate disassembly errors in which IDA Pro fails to identify code bytes immediately following unconditional jumps and returns. These too are a consequence of relying too heavily on control-flow analysis to discover code bytes. Occasionally these errors even appear after conditional jumps, as shown in Case 9. It is unclear why IDA Pro makes this final kind of mistake, though we speculate that it may be the result of a dataflow analysis that incorrectly infers that certain conditional branches are always taken and therefore never fall through. Use of conditional branches as unconditional jumps is a common malware obfuscation technique that this analysis may be intended to counter. However, in this case it backfires and leads to an incorrect disassembly. Our method yields the correct disassembly on the right.

2.5 Graph Based Disassembly Model

Our recent past work is the first to apply machine learning and data mining to address this problem [Wartell et al., 2011]. The approach uses statistical data compression techniques to reveal the semantics of a binary in its assembly form, yielding a segmentation of code bytes into assembly instructions and a differentiation of data bytes from code bytes. Although the technique is effective and exhibits improved accuracy over the best commercial disassembler currently available [Hex-Rays, 2012], the compression algorithm suffers high memory usage and analysis time. Thus, training on large corpora can be very slow compared to other disassemblers.

This section presents an improved disassembly technique that is both more effective and more efficient. Rather than relying on semantic information (which leads to long training times), we leverage a finite state machine (FSM) with transitional probabilities to infer likely execution paths through a sea of bytes. Our main contributions include a graph-based static disassembly technique; a simple, efficient, but effective disassembler implementation; and an empirical demonstration of the effectiveness of the approach.

Our high-level strategy involves two linear passes: a preprocessing step which recovers a conservative superset of potential disassemblies, followed by a filtering step in which a state machine selects the best disassembly from the possible candidates. While the resulting disassembly is not guaranteed to be fully correct (due to the undecidability of the general problem), it is guaranteed to avoid certain common errors that plague mainstream disassemblers. Our empirical analysis shows our simple, linear approach is faster and more accurate than the observably quadratic-time approaches adopted by other disassemblers.

Section 2.5.1 details the solution of the implementation of our FSM disassembler and Section 2.5.6 shows our empirical evaluation of the disassembler's analysis time and mistakes made vs. IDA Pro.

2.5.1 Design

Problem Definition Given an arbitrary string of bytes, what is the most probable execution path through the binary given a corpus of correct binary executable paths?

Our disassembly system consists of a shingled disassembler which allows generation any possible valid execution path, or *shingles*, a finite state machine of trained binary executables, and a graph disassembler that traces and prunes the shingles to output the maximum-likelihood execution path. Figure 2.5 shows the architecture of our disassembly technique.

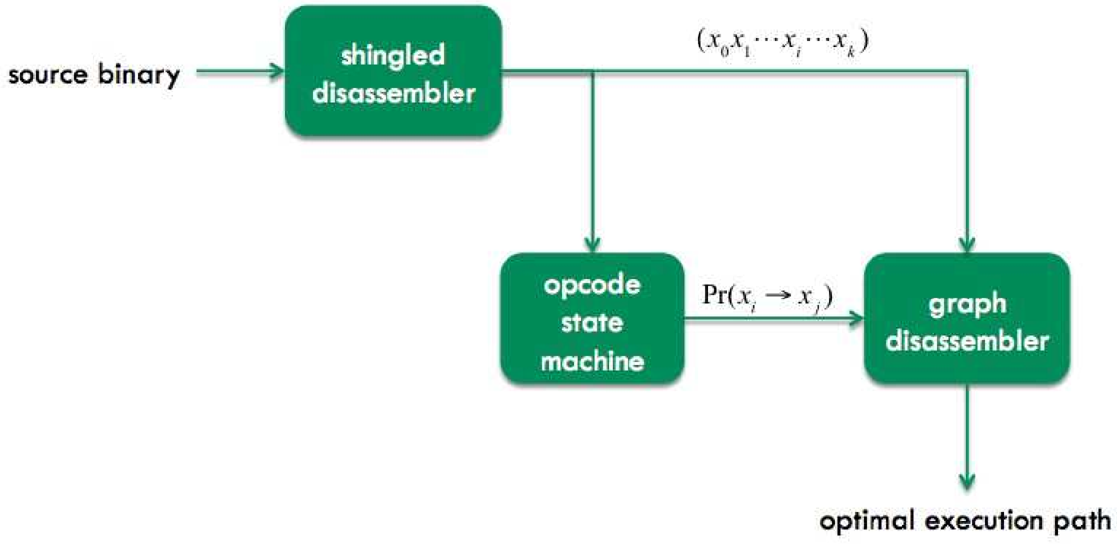


Figure 2.5. Disassembler Architecture

2.5.2 Shingled Disassembler

We use a shingled disassembler described in Section 2.3 as the first pass of our system, effectively storing all possible execution paths. However, during this first initial pass of a code section, more semantic information about a binary can be stored. Local statistics called code/data modifiers that are specific to the executable are collected during this pass. These modifiers keep track of the likelihood that a shingle is code or data in this particular executable. The following heuristics are used to update modifiers:

1. If the shingle at address `i` is a long direct branch instruction with `j` as its target, the address `j` is more likely to be a code instruction. We apply this heuristic with short direct branches as well, but with less weight since two byte instructions are more likely to be seen within other instruction operands.

2. If shingles at addresses `i`, `j` and `k` sequentially fall-through to each other and match one the most common instruction opcode sequences, each of these three addresses is more likely to be code. Common sequences include function prologues, epilogues, etc.

3. If bytes at address `i` and `i + 4` are both treated as addresses, both addresses reference shingles within the code section of the binary, the likelihood that addresses `i` through `i + 7` are data is very high. Shingles `i` through `i+7` are marked as data, as well as any following four byte sequences that match this criteria. This is most likely a series of addresses referenced by a conditional branch elsewhere in the code section.

2.5.3 Opcode State Machine

The state machine is constructed from a large corpus of pre-tagged binaries, disassembled with IDA Pro v6.3. The byte sequences of the training executables are used to build an opcode graph, consisting of opcode states and transitions from one state to another. For each

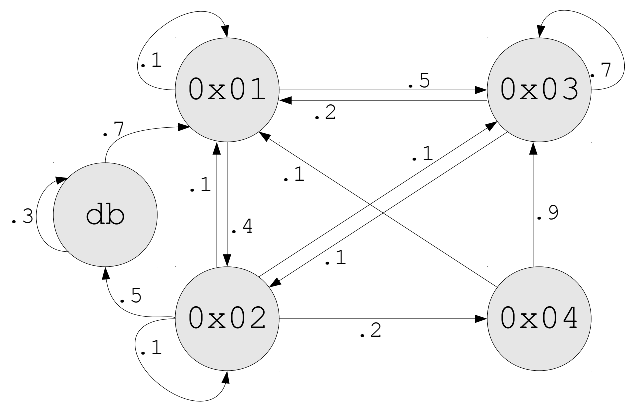


Figure 2.6. Instruction Transition Graph: 4 opcodes

opcode state, we label its transition with the probability of seeing the next opcode in the training instruction streams. The opcode graph is a probabilistic finite state machine (FSM) in thin disguise—essentially a graph including all the correct disassemblies of the training byte sequences annotated with transition probabilities. The accepting state of the FSM is the last unconditional branch seen in the binary.

Figure 2.6 shows what this transition graph might look like if the x86 instruction set only contained four opcodes: 0x01 through 0x04. Each directed edge in the graph between opcode x_i and x_j implies that a transition between x_i and x_j has been observed in the corpus, and the edge weight of $x_i \rightarrow x_j$ is the probability that given x_i, the next instruction is x_j. It is also important to note the node *db* in the graph which represents data bytes. Any transition from an instruction to data observed in the corpus will be represented by a directed edge to the *db* node. The graph for the full x86 instruction set includes more than 500 nodes as each observed opcode must be included.

2.5.4 Maximum-Likelihood Execution Path

We name the output of the shingled disassembler a *shingled binary.* The shingled binary of the source executable can generate a factorial number of valid disassemblies. Our graph disassembler is designed to scan the shingled binary and prune shingles with lower probabilities.

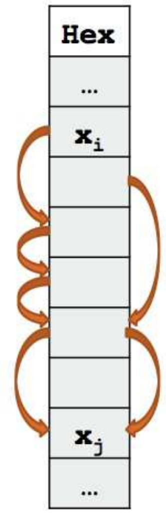


Figure 2.7. Graph disassembly for a shingled binary.

By using our graph disassembler, we can find the maximum-likelihood execution path by tracing the shingled binary through the opcode finite state machine. At every receiving state, we check which preceding path (predecessor) has the highest transition probability. For example in Figure 2.7, address x_j is the receiving state of two preceding addresses. We compute the transition probability from each of the two addresses and sheer the one with a lower probability.

Theorem 2.5.1. *The graph disassembler always returns the maximum-likelihood execution path among all valid shingles $\mathcal{S}$.*

Proof. Each byte in the shingled binary is a potential receiving state of multiple predecessors. At each receiving state, we keep the best predecessor with the highest transition probability. Therefore, when we reach the last receiving state—the accepting state, that is, the last unconditional branch instruction, we find the shingle with the highest probability as the best execution path. □

The transition probability of a predecessor consists of two parts: the global transition probability taken from the opcode state machine and the local modifiers, and local statistics of each byte being code or data based on several heuristics. This is important because runtime reference patterns specific to the binary being disassembled are included in distinguishing the most probable disassembly path.

Let r be a receiving state of a transition triggered at x_i in the shingled binary, $Pr(pred(x_i))$ be the transition probability of the best predecessor of x_i, cm and dm be the code and data modifiers computed during shingled disassembly. The transition probability to r is as follows:

$$Pr(r) = Pr(pred(x_i)) \cdot cm/dm$$

if x_i is a fall-through instruction, or

$$Pr(r) = Pr(pred(x_i)) \cdot cm/dm \cdot Pr(db_i) \cdot Pr(db_r)$$

if x_i is a branch instruction, where $Pr(db_i)$ is the probability that x_i is followed by data and $Pr(db_r)$ is the probability that r is proceeded by data. Every branch instruction can possibly be followed by data. To account for this, when determining the best predecessor for each instruction, branch instructions are treated as fall-throughs to their following instruction and to data. Each branch instruction can be a predecessor to the following instruction or to any instruction that is on a 4-byte boundary and is reachable via data bytes.

Therefore, for any possible valid shingle s resulting a trace of $r_0, \ldots, r_i, \ldots, r_k$, the transition probability of s is:

$$Pr(s) = Pr(r_0)Pr(r_1) \ldots Pr(r_i) \ldots Pr(r_k),$$

and the optimal execution path s^* is:

$$s^* = \arg\max_{s \in \mathcal{S}} Pr(s).$$

2.5.5 Algorithm Analysis

Our disassembly algorithm is much quicker than most other approaches due to the small amount of information that needs to be analyzed. The time complexity of each of the three steps is as follows:

- shingled disassembly: $O(n)$, where n is the number of bytes in the code section;
- sheering: $O(n)$ for pruning invalid shingles;
- graph disassembly: $O(n)$ for a single-pass scanning over the shingled binary.

Therefore, our disassembly algorithm runs in time $O(n)$, that is, linear in the size of the source binary executable.

2.5.6 Evaluation

A prototype of our disassembler was developed in Windows using Microsoft .NET C#. Testing of our disassembly algorithm was done on an Intel Xeon processor with 6 cores at 2.4GHz apiece and 24GB of physical RAM. We tested on a number of different binaries with very positive results.

2.5.7 Broad Results

Table 2.9 shows the different programs on which we tested our disassembler, as well as file sizes and code section sizes. It also displays the number of instructions that the graph disassembler identified that IDA Pro didn't identify as code. Figure 2.8 shows the percentage of instructions that both IDA Pro and our disassembler identified as code.

After the shingled disassembly has been composed, a large number of instructions have already been eliminated from each binary as invalid opcodes or invalid fall-throughs. Figure 2.9 shows the percentage of bytes that have been sheered after the shingled disassembly.

Table 2.9. File Statistics

File Name (.exe)	**File Size (K)**	**Code Size (K)**	**IDA Missed Instr.**
`calc`	114	75	1700
`7z`	163	126	680
`cmd`	389	129	5449
`synergyc`	609	218	12607
`diff`	1161	228	3002
`gcc`	1378	254	2760
`c++`	1380	256	2769
`synergys`	738	319	8061
`size`	1703	581	5540
`ar`	1726	593	8626
`objcopy`	1868	701	6293
`as`	2188	772	7463
`objdump`	2247	780	7159
`steam`	1353	860	16928
`git`	1159	947	9776
`xetex`	14424	1277	18579
`gvim`	1997	1666	19145
`Dooble`	2579	1884	57598
`luatex`	3514	2118	18381
`celestia`	2844	2136	24950
`DosBox`	3727	3013	24217
`emule`	5758	3264	52434
`filezilla`	7994	7085	79367
`IdentityFinder`	23874	12781	180176

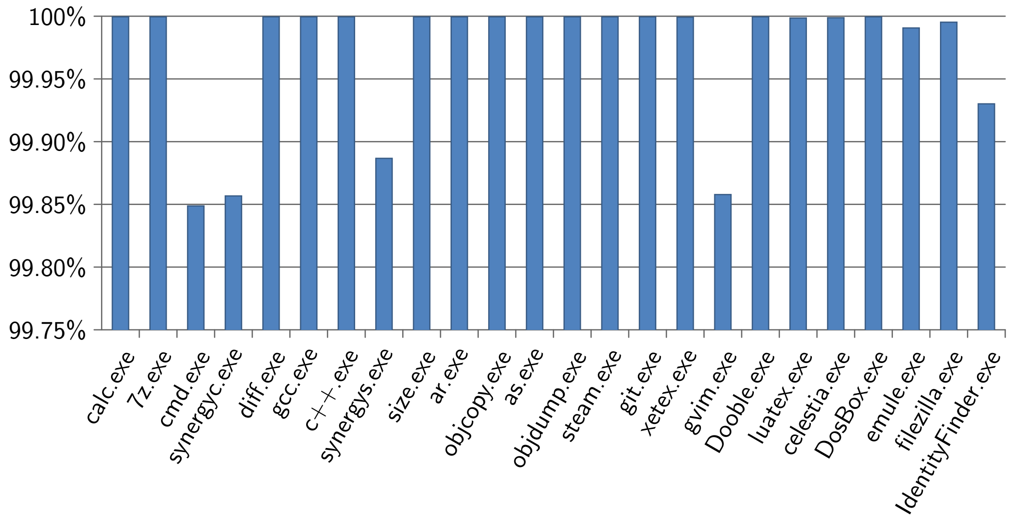


Figure 2.8. Percent of instructions identified by IDA Pro as well as our disassembler.

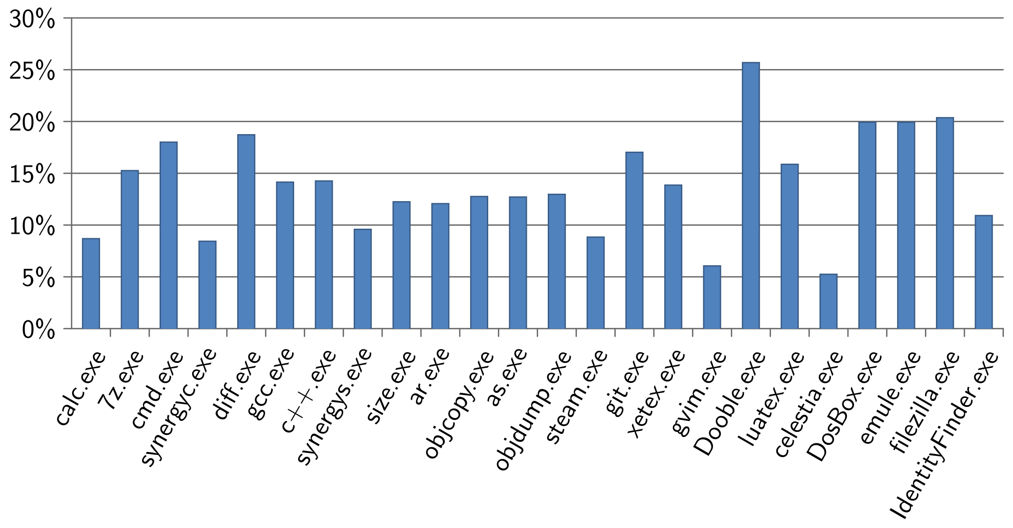


Figure 2.9. Percent of addresses sheered during shingled disassembly.

Our disassembler runs in linear time in the size of the given binary. Figure 2.10 shows how many times longer IDA Pro took to disassemble each binary vs. our disassembler (IDA Time / Our Time). Our disassembler is increasingly faster than IDA Pro as the size of the input grows.

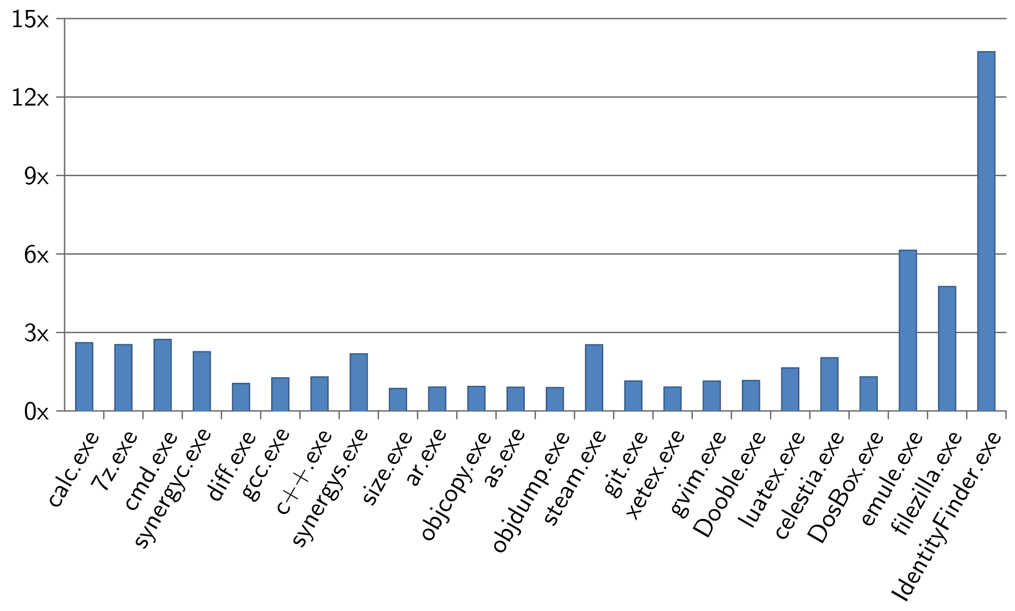


Figure 2.10. Disassembly time vs. IDA Pro

2.5.8 eMule Case Study

The eMule file sharing software is extremely popular, with almost five hundred million downloads on *SourceForge*. It also works well as a case study to compare our disassembler versus IDA Pro to examine some of the mistakes that IDA Pro makes.

We tested IDA Pro v6.3 against our disassembler when working with eMule v.50a. IDA Pro makes a large number of mistakes when attempting to disassemble eMule and ignores vast blocks of code. Our disassembler does not make these mistakes.

The most pressing mistake made by IDA Pro is demonstrated in Case 1, where a large block of instructions are never branched to and do not resemble common code sequences, and are thus classified as data. This problem is so prevalent in eMule that we saw it occur at each of the following addresses `0x524CF0`, `0x5250A0`, `0x525C00`, `0x5262D3`, `0x533090`, `0x62ABBB`, `0x6B2821`, `0x6CF68A`, and `0x711DC9`. More examples of this may exist in eMule, these are merely the addresses that were manually verified by the authors. Our disassembler accurately classifies each of these blocks as code.

Case 2 is an example of IDA Pro using heuristics to determine where data blocks are and mistaking a common opcode sequence for an address in the code section. IDA Pro strictly labels the 4 bytes that look like an address as data, and then mislabels bytes around it to create a normal execution flow. Our disassembler correctly identifies these code bytes.

IDA Pro sometimes drops a single common first byte from an instruction; in Case 3 `0x8B` is dropped and `0xFF` in Case 4. This is an obvious mistake since it is an illegal fall-through; code must fall-through to data if IDA Pro were correct. Our disassembler is incapable of making this mistake based on its architecture.

Cases 5, 6 and 7 are all very similar, each demonstrating IDA Pro's ability to drop a direct branch instruction. Each of these instructions is obviously code; in Case 7 the `jmp` instruction is quite obviously used as a switch statement. Our disassembler doesn't misclassify any of these instructions.

Finally, Case 8 is possibly the most curious of mistakes by IDA Pro since it is an entire function epilogue that is treated as data. Function epilogues are among the most common opcode sequences seen in binaries, so the fact that one is ignored like this is quite strange. Our disassembler correctly identifies it since we use common opcode sequences to help in classification.

Table 2.10. Disassembly Comparison for *eMule.exe*

IDA Pro	**Ours**
Case 1: Large block of code treated as data by IDA Pro (158-1104 bytes)	
41CF9D: call CxxThrowException@8	41CF9D: call CxxThrowException@8
41CFA2: dw 0CC5Bh	41CFA2: pop ebx
...	41CFA3: db align (0CCh x13)
41D030: dd 0CC5B0028h, (0CCh x12)	41CFB0: push 0FFFFFFFFh
41D040: push 0FFFFFFFFh	...
	41D032: pop ebx
	41D033: db align (0CCh x13)
	41D040: push 0FFFFFFFFh
Case 2: Common opcode sequence mistaken for an address (26 bytes)	
41CD3D: call CxxThrowException@8	41CD3D: call CxxThrowException@8
41CD42: db '[????????????d',0	41CD42: pop ebx
41CD53: align 4	41CD43: db align (0CCh x13)
41CD54: dd 548B0000h, 0FF6A0824h	41CD50: mov eax, large fs:0
41CD5C: push offset SEH_41CC30	41CD56: mov edx, [esp+8]
	41CD5A: push 0FFFFFFFFh
	41CD5C: push offset SEH_41CC30
Case 3: 0x8B byte dropped (1 byte)	
525D82: mov edx, [eax+1Ch]	525D82: mov edx, [eax+1Ch]
525D85: db 8Bh	525D85: mov edi, off_7DFAB4
525D86: cmp eax, offset off_7DFAB4	
Case 4: 0xFF byte dropped (1 byte)	
58DC4E: push ecx	58DC4E: push ecx
58DC4F: db 0FFh	58DC4F: call off_7DFAB4
58DC50: adc eax, offset off_7DFAB4	

Table 2.10 continued

IDA Pro	Ours
Case 5: Short direct jmp dropped (2 bytes)	
67882D: call sub_6C978E	67882D: call sub_6C978E
678832: db 0EBh	678832: jmp short loc_678839
678833: db 5	678834: cmp ebp, 0FFFFFFFEh
678834: cmp ebp, 0FFFFFFFEh	
Case 6: Long direct jump dropped (5 bytes)	
71951C: mov ecx, [ebp-10h]	71951C: mov ecx, [ebp-10h]
71951F: db 0E9h	71951F: jmp CWnd@@UAE@XZ
719520: dd 0FFFBA052h	
Case 7: Dropped jump switch statement (14 bytes)	
6C3137: db 83h	6C3137: sub esp, 2Ch
6C3138: dd 0E0832CECh, 8524FF3Fh	6C313A: and eax, 3Fh
6C3140: dd offset off_7DF12E	6C313D: jmp off_7DF12E[eax*4]
6C3144: fdiv st, st	6C3144: fdiv st, st
Case 8: Dropped epilogue (6 bytes)	
6CF231: db 59h	6CF231: pop ecx
6CF232: db 5Fh	6CF232: pop edi
6CF233: db 5Eh	6CF233: pop esi
6CF234: db 0C2h	6CF234: retn 8
6CF235: db 8	
6CF236: db 0	

CHAPTER 3
X86 BINARY REWRITING

Rewriting x86 binaries can be used for optimization, security policy enforcement, binary obfuscation, and other purposes. As was discussed in Chapter 1 however, no system that approaches this problem can provide machine-verifiable safety for legacy COTS binaries wihtout source code or metadata. This chapter will detail the challenges we found in creating such a binary rewriting framework, and the solutions we developed to handle them.

The rest of this chapter is structured as follows: Section 3.1 details some challenges that must be solved by an x86 binary rewriting system. A rewriting system that randomizes basic blocks at runtime to prevent ROP attacks is presented in Section 3.2. A machine verifiable Software Fault Isolation (SFI) and application level security policy enforcement rewriter is detailed in Section 3.3. The intermediary library facilitates system interoperability for both of these systems is presented in Section 3.4.

This work presented in this chapter was published at the 2012 ACM Conference on Computer and Communications Security (CCS) [Wartell et al., 2012b], and the 2012 Annual Computer Security Applications Conference (ACSAC) [Wartell et al., 2012a].

3.1 Rewriting Challenges

Since static x86 disassembly without metadata is an undecidable problem, previous solutions to binary rewriting require code producer cooperation. For example, disassembly is not necessary when using recompilation as a rewriting technique, since instrumentation can be handled as part of compilation. Computed jumps can be omitted from the compiled binary, and any necessary instruction guards can be included and accounted for without issue since all relevant targeting information is available to the compiler.

When rewriting without metadata, much less semantic information is available and thus the rewriting process requires less intuitive solutions. Section 3.1.2 details where we will place our rewritten executable code. How to handle indirect branch instructions is detailed in Section 3.1.3, and successfully hijacking system calls in Section 3.1.4.

3.1.1 Accurate Disassembly

As was discussed in Chapter 2, static x86 disassembly without metadata is provably undecidable. However, in order to correctly rewrite a binary a sound disassembly is needed so that the semantic information in a binary is preserved. If a disassembly is used to rewrite a binary that does not include an execution path X and the rewritten binary attempts to follow X during its execution, the rewritten binary will crash since those instructions will not be included.

To solve this issue, our rewriting scheme takes a conservative disassembly approach. Instead of attempting to find just one disassembly path the executable takes through a code section, our rewriting system includes all possible valid execution paths, as is described by Section 2.3. In order to include all of these paths, shingles that overlap must all be included in the rewritten binary. Figure 2.4 shows an example of a shingled disassembly and the instruction set that results from it.

In order to include a conservative disassembly within the newly rewritten executable, we first include the longest execution path we can find through the executable and include it in the rewritten section. All other overlapping code sections are then added to the end. If they fall-through to an already included execution path, they are concluded with a direct jump instruction, targeting the instruction to which they originally fell through.

This approach to disassembly is conservative, because instructions are included in the rewritten section that will never be executed, but it ensures that no execution paths in the binary can be missed. Since binary rewriting requires all executed instructions to be included

Table 3.1. Inplace Binary rewriting

Original Address	**Original Code**	**Rewritten Address**	**Rewritten Code**
0x004010B0	`mov ebx, [0x004105B4]`	0x004010B0	`mov ebx, [0x004105B4]`
0x004010B6	`call ebx`	0x004010B6	`nop`
		0x004010B7	`call ebx`

in the disassembly, and we cannot statically determine which instructions are executed, this is a conservative approach to obtain a sound disassembly.

3.1.2 Where to rewrite?

The first challenge in rewriting is determining where to place the rewritten instructions. There are two obvious choices: (a) transform the original code in-place, substituting original instruction sequences with new ones, or (b) generate a completely new code section separate from the original.

In-place Rewriting. Rewriting a binary in-place (i.e., instrumenting the code sections of the binary to insert, remove, and modify instructions) would be ideal in terms of efficiency and code bloat. However, any binary rewriter that requires instruction-insertion is difficult, if not impossible, to instrument in place. Rewriting for security purposes typically requires insertion of guard instructions, so in-place rewriting is typically not feasible in such cases. Table 3.1 displays an example of in-place rewriting. The original binary loads an address from the code section in to `ebx`, then branches to the value in `ebx`. However, the inclusion of a single nop instruction adds a single byte to the binary. This shifts the code stored in the binary one byte, causing the information at `0x004105B4` to be different than the original binary, loading the wrong address into `ebx`.

This problem is described in more detail in Section 3.1.3. If it can be somehow surmounted, adding to the size of the executable's code section by injecting instructions is possible, but a

Table 3.2. Pinhole Rewriter

Original Address	**Original Code**	**Rewritten Address**	**Rewritten Code**	**Wrapper Code**
`0x004010B0`	`push ebx`	`0x004010B0`	`push ebx`	*(guard instructions)*
`0x004010B1`	`push ebx`	`0x004010B1`	`push ebx`	`call [0x405014]`
`0x004010B2`	`push ecx`	`0x004010B2`	`push ecx`	`retn`
`0x004010B3`	`call [0x405014]`	`0x004010B3`	`call <Wrapper>`	
`0x004010B9`	`...`	`0x004010B8`	`nop`	
		`0x004010B9`	`...`	

problem immediately arises. Binary sections each have a specific amount of virtual space assigned in the section header, so if that is exceeded, the binary will fail to execute. If the executable section is the last section, the section size simply needs to be increased. However, most compilers do not place the executable section last, and thus increasing the size of the section will cause it to overlap other sections in virtual address space. To our knowledge, no rewriter actually attempts this approach.

There are forms of rewriting in-place that do not suffer from this problem, such as pinhole rewriting. A pinhole rewriter does not actually inject guard instructions, instead replacing the guarded instruction with a direct call to a wrapper function containing the intended guard instructions. Table 3.2 shows an example of a pinhole rewriter that guards dereferenced branch instructions.

However, these rewriting techniques require that instructions requiring guards be replaceable. For example, a simple `call eax` is only two bytes, and replacing such an instruction with a branch to a wrapper function is infeasible. A direct branch instruction is at least five bytes, so the previous pinhole approach does not work. Most interesting rewriters are not implementable as a pinhole rewriter for this reason.

Binary rewriters can also be used to perform optimizations. A rewriter that replaces target instruction sequences with shorter ones can be easily implemented in-place. However, optimizations are a small subset of the tasks performed by rewriters, and often take place in addition to other rewriting techniques. Etch [Romer et al., 1997] is one example of such a rewriting system that allows for pin-holes or optimizations.

Adding a new section. Rather than attempt to rewrite in-place, we can create a new section after the already existing sections in the executable, since doing so will not cause any harm to the executable. Then, we can include all the information from the section we want to rewrite, along with any modifications the rewriter wishes to include. However, to include these modifications, the rewriter must be able to distinguish the intended semantics of the executable section. As described in Chapter 2, this problem is far from trivial.

Since code and data is interleaved inside of a code section, we can leave the original code section non-executable and treat it as a data section. Thus all data in it is maintained and still accessible, effectively separating the code and data of an x86 binary into separate sections. This allows all instructions that reference data in the binary to be left unmodified, since they will be referencing the old code section. Throughout the rest of this dissertation, the code section of the original binary will be referred to as the `.text` section, the old code section that is now non-executable in the rewritten binary will be referred to as the `.told` section, and the new executable section will be referred to as the `.tnew` section.

After the rewriter has created sections `.told` and `.tnew`, it creates a copy of the original binary, injects `.tnew`, overwrites `.text` with `.told`, and modifies header information appropriately to ensure proper execution. The new binary is then ready for for propagation.

3.1.3 Control Flow Instructions

Direct Branches. All instructions that that do not always fall-through must be modified or guarded in order to ensure proper execution. First, all direct `jmp`, `call` and `jcc` instructions must be modified. Since no instruction is in its original position, the intended target of

Table 3.3. Direct Jump Modification Example

Original code	**Rewritten code**
`0x004010A0:  cmp eax, ebx`	`0x005021A0:  cmp eax, ebx`
`0x004010A3:  jne 10h ; Loc_1`	`0x00503BC3:  jne 1A23h ; New_Loc_1`
`...`	`...`
`Loc_1:`	`New_Loc_1:`
`0x004010B3:  pop ebx`	`0x00503BC6:  pop ebx`

the direct branch must be determined and maintained after instrumentation. This involves modifying the offset of these instructions after their positions in the new code section have been determined.

Table 3.3 shows examples of modified direct branches in the `.tnew` section. Short direct branch instructions are conservatively converted into their longer encodings (changing them from 2 byte instructions to 6 byte instructions). This is necessary in systems like Stir as described in Section 3.2, however for basic rewriters this is not necessary if the branch instruction and its target have not separated by more than 127 bytes. Such optimizations can be implemented discretionally.

Indirect Branches. Indirect branch instructions are more difficult to handle, however, since their intended target is computed at runtime. It is provably undecidable to statically determine the targets of indirect branches in general. Furthermore, indirect branches come in many varieties as was previously shown in Table 2.2.

The diversity of indirect branch techniques allow the target of an indirect branch to be determined from an address stored in the code or data sections, an address in a register, or dereferencing a jump table using a register as the index. For the sake of simplicity and minimal overhead, a solution that encompasses the redirection of all 4 of these branch types to their new targets is ideal.

Table 3.4. Summary of x86 code transformations

Description	Original code	Rewritten code

Computed jumps with register operands	`call ebx`	`cmp byte ptr [ebx], 0xF4`
		`cmovz ebx, [ebx+1]`
		`call ebx`

Computed jumps with memory operands	`call [0x00483BEC]`	`mov eax, [0x00483BEC]`
		`cmp byte ptr [eax], 0xF4`
		`cmovz eax, [eax+1]`
		`call eax`

While computed jumps may seem rare to those accustomed to source-level programming, they pervade almost all binary programs compiled from all source languages. Computed jumps 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.

When an indirect branch is encountered, if it points to the `.told`, we must redirect it to the corresponding address in `.tnew`. However, if it points to `.tnew`, there is no need to modify the address. This is solved by writing a tag byte and the new address of the target over the first 5 bytes at their old code address. Table 3.4 displays the transformation that is possible due to this system.

Since the transformed binary stores a jump table of addresses in its old code section, indirect branches need only check whether their target is in `.told` (i.e. the first byte is `0xF4`), and if so, dereference the address. Otherwise, the target is in the `.tnew` and does not need to be dereferenced.

Table 3.4 shows modifications involving a scratch register (`eax` in the example). Use of a scratch register isn't necessary, however. These instructions can be modified to instead store the address above the stack (e.g. `mov [esp-4], ebx`), ensuring that no register values are corrupted at runtime. Each of these solutions results in proper execution; however, since using a scratch register results in less guard instructions we expect slightly faster runtimes.

Jump Table Examples. To illustrate how indirect branches are handled, whether their targets reference the `.told` or somewhere else in memory, Figures 3.1 and 3.2 demonstrate the transformation process for two representative assembly codes.

Figure 3.1 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. The same `mov` instruction is present in our rewritten version, however the call instruction is replaced with the guarded call sequence shown in lines 2–4 of the rewritten binary. The compare (`cmp`) and conditional move (`cmovz`) implement the table-lookup, but the address in `esi` is an imported function, and thus does not begin with `0xF4`. The `cmovz` instruction does not dereference the address and the ensuing call executes correctly.

Figure 3.2 shows a computed jump with a memory operand that indexes the jump table residing in `.told`. The rewritten code first loads the destination address into a scratch register (`eax`) in accordance with row 2 of Table 3.4. It then implements the same lookup as in Fig. 3.1. This time the lookup has a significant effect—it discovers at runtime that the address drawn from the lookup table must be retargeted 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.

3.1.4 Hijacking System Calls

The transformations described in Section 3.1.3 would be enough to result in working, instrumented binaries, except that x86 binaries have other intricacies that must be solved. Hidden entry points, known as *callbacks*, are prevalent in x86 binaries. A callback occurs when the executing binary calls a library function with an address in its code section as an argument. That argument is later used to branch back into the binary from the library. After instrumentation, this becomes an issue because this will be an unguarded branch that

Original:
```
.text:00499345 8B 35 FC B5 4D 00 mov esi, [4DB5FCh] ;IAT:MBTWC
...
.text:00499366 FF D6             call esi
```
Rewritten:
```
.text:0059DBF0 8B 35 FC B5 4D 00 mov esi, [4DB5FCh] ;IAT:MBTWC
...
.tnew:0059DC15 80 3E F4          cmp byte ptr [esi], F4h
.tnew:0059DC18 0F 44 76 01       cmovz esi, [esi+1]
.text:0059DC1C FF D6             call esi
```

Figure 3.1. 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:
```
.told: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 [esi], F4h
.tnew:004F33BE 0F 44 40 01          cmovz eax, [eax+1]
.tnew:004F33C2 FF E0                jmp eax
...
.tnew:004F3A60 3D 8C 8A 4D 00       cmp byte_4D8A8C, 0
.tnew:004F3A67 74 27                jz loc_4F3A90
.tnew:004F3A69 84 C9                test cl, cl
.tnew:004F3A6B 74 22                jz short loc_4F3A90
```

Figure 3.2. Rewriting code that uses a jump table

refers to `.told`. This may seem like an uncommon problem, but it actually manifests in almost every binary. A simple GCC compiled program with an empty main function has two callbacks at the binary level, one for initialization and one for termination.

There are two solutions to this problem—rewriting every library to which the instrumented binary refers or catching the callback and replacing its argument at runtime. The first solution is impractical in most cases since it would involve rewriting `kernel32.dll` and other libraries that include trusted, unrewritable control-flows, such as direct kernal traps. Redirecting all callbacks raises its own challenges however.

A common solution to such a problem is the use of *Import Address Table (IAT) hooking*, wherein the addresses in the IAT are replaced with the addresses of wrapper functions. However, in a security setting this solution is too easily subverted. For example, a binary can implement the functionality of `GetProcAddress` internally to fetch the address of a system function, or by simply guessing the address of the system function. Both of these methods circumvent IAT hooking, since the system function being used is never stored in the IAT.

Instead, we use a more thorough technique that identifies code points where an IAT address is called, an IAT address is loaded into a register, or a library function is dynamically loaded. Each of these occurrences must be properly handled, lest the sandboxing guards redirect the flows back into the old code section, corrupting the library call.

Table 3.5 displays the code transformation that handles callbacks. Calls to callback registration functions are redirected to an intermediary library with the address of the callback registration function as a parameter. Based on the signature of the call, the parameter that points to the old code section is checked for an `0xF4` byte, and dereferenced if one is found. A more thorough explanation of this process is described in Section 3.4.1.

Table 3.5. Summary of x86 code transformations

Description	**Original code**	**Rewritten code**
IAT loads	`mov eax, [.idata:printf]`	`mov eax, offset trampoline_printf`
		`trampoline_printf:`
		`and [esp],` *0x00FFFFF0*
		`jmp [.idata:printf]`
Callback registrations	`call [.idata:atexit]`	`jmp trampoline_atexit`
		`trampoline_atexit:`
		`push [.idata:atexit]`
		`call intermediary.reg_callback`
		`return_trampoline:`
		`call intermediary.callback_ret`

3.2 STIR

All of the modifications to a binary necessary to make it viable for rewriting are detailed in Section 3.1, but this is only the first step. The next step is to create interesting rewriters using these techniques to accomplish interesting changes to binaries to implement interesting security policies.

New binary level attacks have surfaced recently that use user-level code within the binary to manifest a malicious action. Such attacks rely on modifying the stack at runtime to modify the targets of `retn` instructions within the binary and chain *gadgets* in the user–level code. These gadgets, when executed in sequence create a malicious action.

This section introduces a new technique, *Self-Transforming Instruction Relocation (*Stir*)*, that transforms legacy application binary code into *self-randomizing* code that statically re-randomizes itself each time it is loaded. The capacity to re-randomize legacy code (i.e., code without debug symbols or relocation information) at load-time greatly eases deployment, and its static code transformation approach yields significantly reduced performance overheads. Moreover, randomizing at basic block granularity achieves higher entropy than Address Space Layout Randomization (ASLR) [PaX Team, 2003], which only randomizes section base addresses, and can therefore be susceptible to derandomization attacks [Shacham et al., 2004, Roglia et al., 2009].

Stir is a fully automatic, binary-centric solution that does not require any source code or symbolic information for the target binary program. Stir-enabled code randomly reorders the basic blocks in each binary code section each time it is launched, frustrating attempts to predict the locations of gadgets. It is therefore fully transparent, and there is no modification to the OS or compiler. This makes it easily deployable; software vendors or end users need only apply Stir to their binaries to generate one self-randomizing copy, and can thereafter distribute the binary code normally.

Randomizing legacy CISC code for real-world OS's (Microsoft Windows and Linux) without compiler support raises many challenges, including semantic preservation of dynamically computed jumps, code interleaved with data, function callbacks, and imperfect disassembly information. These challenges are detailed further in §3.1. In this chapter, we develop a suite of novel techniques, including conservative disassembly, jump table recovery, and dynamic dispatch, to address these challenges. 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 a potential instruction starting point and static data, as we described in Section 3.1.1. This obviates the need for perfect disassemblies, which are seldom realizable in practice without source code.

Stir was published in the 2012 ACM Conference on Computer and Communications Security [Wartell et al., 2012b].

The rest of this section is structured as follows. Section 3.2.1 details the type of attack we are attempting to prevent. The design and evaluation of the STIR architecture are detailed in Sections 3.2.2 and 3.2.3.

3.2.1 ROP Protection

Subverting control-flows of vulnerable programs by hijacking function pointers (e.g., return addresses) and redirecting them to shell code has long been a dream goal of attackers. For such an attack to succeed, there are two conditions: (1) the targeted software is vulnerable to redirection, and (2) the attacker-supplied shell code is executable. Consequently, to stop these attacks, a great deal of research has focused on identifying and eliminating software vulnerabilities, either through static analysis of program source code (e.g., [Larochelle and Evans, 2001]) or through dynamic analysis or symbolic execution of program binary code (e.g., [Cadar et al., 2006, Godefroid et al., 2008]).

Meanwhile, there is also a significant amount of work focusing on how to prevent the execution of shell code based on its origin or location. Initially, attackers directly injected malicious machine code into vulnerable programs, prompting the development of W$\oplus$X (*write-xor-execute*) protections such as DEP [Andersen, 2004] and ExecShield [van de Ven, 2004] to block execution of the injected payloads. In response, attackers began to redirect control flows directly to potentially dangerous code already present in victim process address spaces (e.g., in standard libraries), bypassing W$\oplus$X. Return-into-libc attacks [Solar Designer, 1997] and return oriented programming (ROP) [Shacham, 2007, Buchanan et al., 2008, Checkoway et al., 2010] are two major categories of such attacks. As a result, address space layout randomization (ASLR) [PaX Team, 2003, Bhatkar et al., 2005] was invented to frustrate attacks that bypass W$\oplus$X.

ASLR has significantly raised the bar for standard library-based shell code because attackers cannot predict the addresses of dangerous instructions to which they wish to transfer control. However, a recent attack from Q [Schwartz et al., 2011] has demonstrated that attackers can alternatively redirect control to shell code constructed from *gadgets* (i.e., short instruction sequences) already present in the application binary code. Such an attack is extremely dangerous since instruction addresses in most application binaries are fixed (i.e., static) once compiled (except for position independent code). This allows attackers to create robust shell code for many binaries [Schwartz et al., 2011].

Recent attempts to solve this issue have employed both static and dynamic techniques. In-place-randomization (IPR) [Pappas et al., 2012] statically smashes unwanted gadgets by

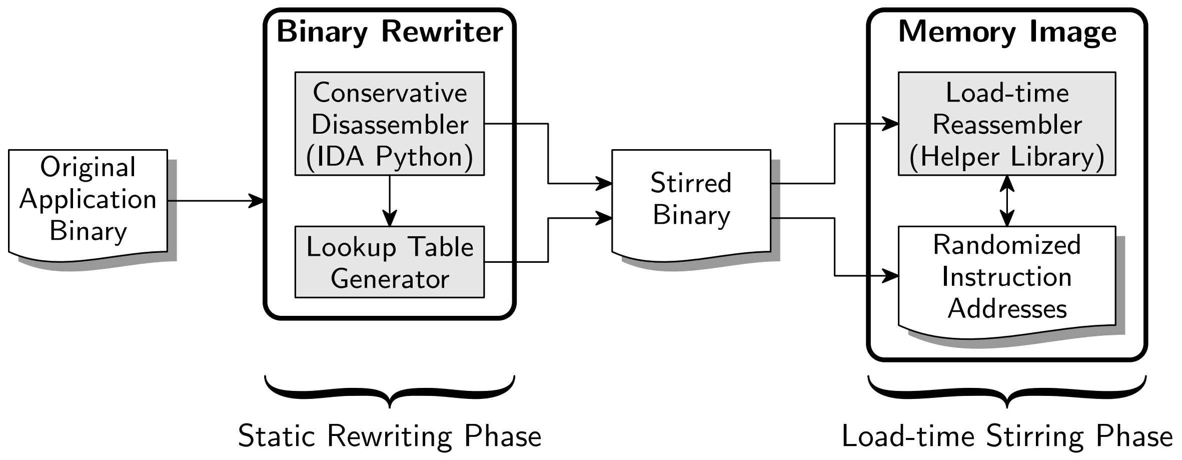

Binary Rewriter
Conservative Disassembler (IDA Python)
Lookup Table Generator
Original Application Binary
Stirred Binary
Memory Image
Load-time Reassembler (Helper Library)
Randomized Instruction Addresses
Static Rewriting Phase
Load-time Stirring Phase

Figure 3.3. System architecture

changing their semantics or reordering their constituent instructions without perturbing the rest of the binary. Alternatively, ILR [Hiser et al., 2012] dynamically eliminates gadgets by randomizing all instruction addresses and using a fall-through map to dynamically guide execution through the reordered instructions. While these two approaches are valuable first steps, IPR suffers from deployment issues (since millions of separately shipped, randomized copies are required to obtain a sufficiently diverse field of application instances), and ILR suffers from high performance overhead (because of its highly dynamic, VM-based approach).

3.2.2 Design

The architecture of STIR is shown in Fig. 3.3. It includes three main components: (1) a conservative disassembler, (2) a *lookup table generator*, and (3) a load-time reassembler. At a high level, our disassembler takes a target binary and transforms it to a randomizable representation. An address map of the randomizable representation is encoded into the new binary by the lookup table generator. This is used by the load-time reassembler to efficiently randomize the new binary's code section each time it is launched.

This section presents a detailed design of each component. We first outline the static phase of our algorithm (our conservative disassembler and lookup table generator) in §3.2.2,

Algorithm 3: *Trans*(α, c): Translate one instruction

```
Input: address mapping $\alpha : \mathbb{Z} \rightharpoonup \mathbb{Z}$ and instruction $c$
Output: translated instruction(s)
  if $IsComputedJump(c)$ then
    $op \leftarrow Operand(c)$
    if $IsRegister(op)$ then
      return [ cmp $op$, F4h;
               cmovz $op$, [$op$+1]; $c$ ]
    else if $IsMemory(op)$ then
      $Operand(c) \leftarrow$ eax
      return [ mov eax, $op$;
               cmp [eax], F4h;
               cmovz eax, [eax+1]; $c$ ]
    end if
  else if $IsDirectJump(c)$ then
    $t \leftarrow OffsetOperand(c)$
    return $c$ with operand changed to $\alpha(t)$
  else
    return $c$
  end if
```

followed by the load-time phase (our reassembler) in §3.2.2. Section 3.2.2 walks through an example. Finally, §3.2.2 addresses practical compatibility issues.

Static Rewriting Phase. Target binaries are first disassembled to assembly code. We use the IDA Pro disassembler from Hex-rays for this purpose [Hex-Rays, 2012], though any disassembler capable of accurately identifying likely computed jump targets could be substituted.

The resulting disassembly may contain harmless errors that misidentify data bytes as code, or that misidentify some code addresses as possible computed jump targets; but errors that omit code or misidentify data as computed jump targets can lead to non-functional rewritten code. We therefore use settings that encourage the disassembler to interpret all bytes that constitute valid instruction encodings as code, and that identify all instructions that implement prologues for known calling conventions as possible computed jump targets. This approach is detailed in Section 3.1.1. These settings suffice to avoid all harmful disassembly errors in our experiments (see §3.2.3).

The assembly code is next partitioned into basic blocks, where a basic block can be any contiguous sequence of instructions with a single entry point. Each block must also end with an unconditional jump, but STIR can meet this requirement by inserting `jmp 0` instructions (a semantic no-op) to partition the code into arbitrarily small blocks during rewriting. The resulting blocks are copied and translated into a new binary section according to Algorithms 3–4, which we implemented as an IDAPython script.

Algorithm 3 translates a single instruction into its replacement in the new code section. Most instructions are left unchanged, but computed jumps are replaced with the lookup table code described in §3.1.3, and direct branches are re-pointed according to address mapping α.

Algorithm 4 calls Algorithm 3 as a subroutine to translate all the instructions. Its initial pass first computes mapping α by using identity function ι as the address mapping.[1] The second pass uses the resulting α to generate the final new code section with direct branches re-targeted.

Once the new code section has been generated, the lookup table generator overwrites all potential computed jump targets t in the original code section with a tag byte `0xF4` followed by 4-byte pointer $\alpha(t)$. This implements the lookup table described in §3.1.3.

It may seem more natural to implement range checks to identify stale pointers rather than using tag bytes. However, in general a stirred binary may consist of many separate modules, each of which has undergone separate stirring, and which freely exchange stale code pointers at runtime. Since each module loads into a contiguous virtual address space, it is not possible to place all the old code sections within a single virtual address range. Thus, implementing pointer range checks properly would require many nested conditionals, impairing performance.

[1]Some x86 instructions' lengths can change when ι is replaced by α. Our rewriter conservatively translates these to their longest encodings during the first pass to avoid such changes, but a more optimal rewriter could use multiple passes to generate smaller code.

Algorithm 4: Translate all instructions

```
Input: instruction list $C$
Output: rewritten block list $B$
  $B \leftarrow [\,]$
  $\alpha \leftarrow \emptyset$
  $t \leftarrow$ base address of .told section
  $t' \leftarrow$ base address of .tnew section
  for all $c \in C$ do
    if $\mathit{IsCode}(c)$ then
      $\alpha \leftarrow \alpha \cup \{(t, t')\}$
      $t' \leftarrow t' + |\mathit{Trans}(\iota, c)|$
    end if
    $t \leftarrow t + |c|$
  end for
  for all $c \in C$ do
    if $\mathit{IsCode}(c)$ then
      append $\mathit{Trans}(\alpha, c)$ to $B$
    end if
  end for
  return $B$
```

Our use of tag bytes reduces this to a single conditional move instruction and no conditional branches.

The resulting binary is finalized by editing its binary header to import the library that performs binary stirring at program start. PE and ELF headers cannot be safely lengthened without potentially moving the sections that follow, introducing a host of data relocation problems. To avoid this, we simply substitute the import table entry for a standard system library (`kernel32.dll` on Windows) with an equal-length entry for our library. Our library exports all symbols of the system library as forwards to the real system library, allowing it to be transparently used as its replacement. This keeps the header length invariant while importing all the new code necessary for stirring.

Load-time Stirring Phase. When the rewritten program is launched, the STIR library's initializer code runs to completion before any code in STIR-enabled modules that link to it.

On Windows this is achieved by the system load order, which guarantees that statically linked libraries initialize before modules that link to them. On Linux, the library is implemented as a shared object (SO) that is injected into the address space of STIR-enabled processes using the LD_PRELOAD environment variable. When this variable is set to the path of a shared object, the system loader ensures that the shared object is loaded first, before any of the other libraries that a binary may need.

The library initializer performs two main tasks at program start:

1. All basic blocks in the linking module's `.tnew` section are randomly reordered. During this stirring, direct branch operands are retargeted according to address mapping α, computed during the static phase.

2. The lookup table in the linking module's `.told` section is updated according to α to point to the new basic block locations.

Once the initialization is complete, the `.tnew` section is assigned the same access permissions as the original program's `.text` section. This preserves non-writability of code employing W$\oplus$X protections.

To further minimize the attack surface, the library is designed to have as few return instructions as possible. The majority of the library that implements stirring is loaded dynamically into the address space at library initialization and then unloaded before the stirred binary runs. Thus, it is completely unavailable to attackers. The remainder of the library that stays resident performs small bookkeeping operations, such as callback support (see §3.2.2). It contains less than 5 return instructions total.

An Example. To illustrate our technique, Fig. 3.4 shows the disassembly of a part of the original binary's `.text` section and its counterparts in the rewritten binary's `.told` and `.tnew` sections after passing through the static and load-time phases described above.

The disassembly of the `.text` section shows two potential computed jump targets, at addresses `0x404B00` and `0x404B18`, that each correspond to a basic block entry point. In the rewritten `.told` section, the underlined values show how each is overwritten with the tag byte `0xF4` followed by the 4-byte pointer $\alpha(t)$ that represents its new location in the `.tnew` section.[2] All remaining bytes from the original code section are left unchanged (though the section is set non-executable) to ensure that any data misclassified as code is still accessible to instructions that may refer to it.

The `.tnew` section contains the duplicated code after stirring. Basic blocks `0x404B00`, `0x404B10` and `0x404B18`, which were previously adjacent, are relocated to randomly chosen positions `0x51234C`, `0x53AF21` and `0x525B12` (respectively) within the new code section. Non-branch instructions are duplicated as is, but static branches are re-targeted to the new locations of their destinations. Additionally, as address `0x525B16` shows, branch instructions are conservatively translated to their longest encodings to accommodate their more distant new targets.

Special Cases. Real-world x86 COTS binaries generated by arbitrary compilers have some obscure features, some of which required us to implement special extensions to our framework to support them. In this section we describe the major ones and our solutions.

Callbacks.

Real-world OS's—especially Windows—make copious use of callbacks for event-driven programming. User code solicits callbacks by passing code pointers to a *callback registration function* exported by the system. The supplied pointers are later invoked by the OS in response to events of interest, such as mouse clicks or timer interrupts. Our approach of

[2]This value changes during each load-time stirring.

Original .text:

```
.text:00404AF0  00 4B 40 00           .dword 00404B00h
.text:00404AF4  18 4B 40 00           .dword 00404B18h
.text:00404AF8  CC (×8)               .align 16
.text:00404B00  8B 04 85 F0 4A 40 00  mov eax,[eax*4+404AF0h]
.text:00404B07  FF E1                 jmp eax
.text:00404B09  CC CC CC CC CC CC CC  .align 16
.text:00404B10  55                    push ebp
.text:00404B11  8B E5                 mov esp, ebp
.text:00404B13  C3                    retn
.text:00404B14  CC CC CC CC           .align 8
.text:00404B18  55                    push ebp
.text:00404B19  83 F8 01              cmp eax, 1
.text:00404B1C  7D 02                 jge 404B20h
.text:00404B1E  33 C0                 xor eax, eax
.text:00404B20  8B C1                 mov eax, ecx
.text:00404B22  E8 D9 FF FF FF        call 404B00h
```

STIRred .told (Jump Table):

```
.told:00404AF0  00 4B 40 00 18 4B 40 00
.told:00404AF8  CC CC CC CC CC CC CC CC
.told:00404B00  F4 4C 23 51 00 40 00 FF
.told:00404B08  E1 CC CC CC CC CC CC CC
.told:00404B10  55 8B E5 C3 CC CC CC CC
.told:00404B18  F4 12 5B 52 00 02 33 C0
.told:00404B20  8B C1 E8 D9 FF FF FF
```

STIRred .tnew:

```
.tnew:0051234C  8B 04 85 F0 4A 40 00  mov eax,[eax*4+404AF0h]
.tnew:00512353  80 38 F4              cmp F4h, [eax]
.tnew:00512356  0F 44 40 01           cmov eax, [eax+1]
.tnew:0051235A  FF E1                 jmp eax
                  ... (other basic blocks) ...
.tnew:00525B12  55                    push ebp
.tnew:00525B13  83 F8 01              cmp eax, 1
.tnew:00525B16  0F 8D 00 00 00 02     jge 525B1Eh
.tnew:00525B1C  33 C0                 xor eax, eax
.tnew:00525B1E  8B C1                 mov eax, ecx
.tnew:00525B20  E8 27 C8 FE FF        call 51234C
                  ... (other basic blocks) ...
.tnew:0053AF21  55                    push ebp
.tnew:0053AF22  8B E5                 mov esp, ebp
.tnew:0053AF24  C3                    retn
```

Figure 3.4. A stirring example

dynamically re-pointing stale pointers at the sites of dynamic calls does not work when the call site is located within an unstirred binary, such as an OS kernel module.

To compensate, our helper library hooks [Hoglund and Butler, 2006] all import address table entries of known callback registration functions exported by unstirred modules. The hooks re-point all calls to these functions to a helper library that first identifies and corrects any stale pointer arguments before passing control on to the system function. This interposition ensures that the OS receives correct pointer arguments that do not point into the old code section.

The full implementation details of this are described in Section 3.4.1.

Position Independent Code

PIC instructions compute their own address at runtime and perform pointer arithmetic to locate other instructions and data tables within the section. An underlying assumption behind this implementation is that even though the absolute positions of these instructions in the virtual address space may change, their position relative to one another does not. This assumption is violated by stirring, necessitating a specialized solution.

All PIC that we encountered in our experiments had the form shown in the first half of Fig. 3.5. The call instruction has the effect of pushing the address of the following instruction onto the stack and falling through to it. The following instruction pops this address into a register, thereby computing its own address. Later this address flows into a computation that uses it to find the base address of a global offset table at the end of the section. In the example, constant `56A4h` is the compile-time distance from the beginning of the pop instruction to the start of the global offset table.

To support PIC, our rewriter identifies call instructions with operands of 0 and performs a simple data-flow analysis to identify instructions that use the pushed address in an arithmetic computation. It then replaces the computation with an instruction sequence of the same

Original:
```
.text:0804894B  E8 00 00 00 00    call 08048950h
.text:08048950  5B                pop ebx
.text:08048951  81 C3 A4 56 00 00 add ebx, 56A4h
.text:08048957  8B 93 F8 FF FF FF mov edx, [ebx-8]
```

Rewritten:
```
.tnew:0804F007  E8 00 00 00 00    call 0804F00Ch
.tnew:0804F00C  5B                pop ebx
.tnew:0804F00D  BB F4 DF 04 08    mov ebx, 0804DFF4h
.tnew:0804F012  90                nop
.tnew:0804F013  8B 93 F8 FF FF FF mov edx, [ebx-8]
```

Figure 3.5. Position-independent code

length that loads the desired address from the STIR system tables. This allows the STIR system to maintain position independence of the code across stirring. In Fig. 3.5, the `nop` instruction is added to ensure that the length matches that of the replaced computation.

Our analysis is not guaranteed to find all possible forms of PIC. For example, PIC that uses some other instruction to compute its address, or that allows the resulting address to flow through the heap before use, would defeat our analysis, causing the rewritten binary to crash at runtime. However, our analysis sufficed to support all PIC instances that we encountered, and compiler documentation of PIC standards indicates that there is only a very limited range of PIC implementations that needs to be supported [Oracle Corporation, 2010].

Statically Computed Returns

Although returns are technically computed jumps (because they draw their destinations from the stack), our rewriting algorithm does not guard them with checks for stale pointers. This is a performance optimization that assumes that all return addresses that we wish to preserve (i.e., those not introduced by an attacker) are pushed onto the stack by calls; thus, no return addresses are stale.

```
              Original .text:                        Jump table .told:
             func_1:                                 func_1:
             .text:40EAA9 33 C0  xor eax, eax        .text:40EAA9 F4 2E
2 bytes lost .text:40EAAB C3     retn                .text:40EAAB 04
             func_2:                                 func_2:
             .text:40EAAC 50     push eax            .text:40EAAC F4

             func_1:                                 func_1:
             .text:40EAA9 33 C0  xor eax, eax        .text:40EAA9 F4 2E
             .text:40EAAB 5B     pop ebx             .text:40EAAB 25
0 bytes lost .text:40EAAC 5E     pop esi             .text:40EAAC 42
             .text:40EAAD C3     retn                .text:40EAAD 00
             func_2:                                 func_2:
             .text:40EAAE 50     push eax            .text:40EAAE F4
```

Figure 3.6. Overlapping function pointers

This assumption was met by all binaries we studied except for a certain pattern of initializer code generated by GNU Compilers. The code sequence in question pushes three immediate operands onto the stack, which later flow to returns. We supported this by treating those three instructions as a special case, augmenting them with stale pointer checks that correct them at the time they are pushed instead of at the time they are used. A more general solution could rewrite all return instructions with stale pointer guards, probably at the cost of performance.

Short Functions

Our jump table implementation overwrites each computed jump target with a 5-byte tagged pointer. This design assumes that nearby computed jump targets are at least 5 bytes apart; otherwise the two pointers must overlap. An example of this type of jump table collision is shown in Fig. 3.6, where the first row has two jump table destinations overlapping two bytes of each other, and the second row does not overlap at all. Such closely packed destinations are rare, since most computed jump destinations are already 16-byte aligned for performance reasons, and since all binaries compatible with *hot-patching* technology have at least 5 bytes of padding between consecutive function entry points (enough to encode a long jump instruction) [Microsoft Corporation, 2005].

In the rare case that two computed jump targets are closer, the rewriter strategically chooses stirred block addresses within the new code section whose pointer representations can safely overlap. For example, if the `.tnew` section is based at address `0x04000000`, the byte sequence `F4 00 F4 00 04 00 04` encodes two overlapping, little-endian, tagged pointers to basic block addresses `0x0400F400` and `0x04000400`, respectively. This strategy suffices to support at least 135 two-pointer collisions and 9 three-pointer collisions per rewritten code page—far more than we saw in any binary we studied.

3.2.3 Evaluation

We have implemented STIR and evaluated it on both Windows and Linux platforms with a large number of legacy binary programs. Our experimental results show that STIR can successfully transform application binaries with self-randomized instruction addresses, and that doing so introduces about 2.4% overhead (significantly better than ILR's 16% [Hiser et al., 2012]) on average at runtime to the applications.

Rewriting Time and Space Overheads

To evaluate the effectiveness of our system, we tested both the Windows and Linux versions of STIR with a variety of COTS and benchmark binaries. Both Windows and Linux tests were carried out on Windows 7 and Ubuntu 12 running on an Intel Core i5 dual core, 2.67GHz laptop with 4GB of physical RAM.

On Windows, we tested STIR against the SPEC CPU 2000 benchmark suite as well as popular applications like Notepad++ and DosBox. For the Linux version, we evaluated our system against the 99 binaries in the coreutils tool-chain (v7.0) for the Linux version. Due to space limitations, figures only present Windows binaries and a selection of 10 Linux binaries. In all of our tests, stirred binaries exhibited the same behavior and output as their original counterparts. Average overheads only cover binaries that run for more than 500ms.

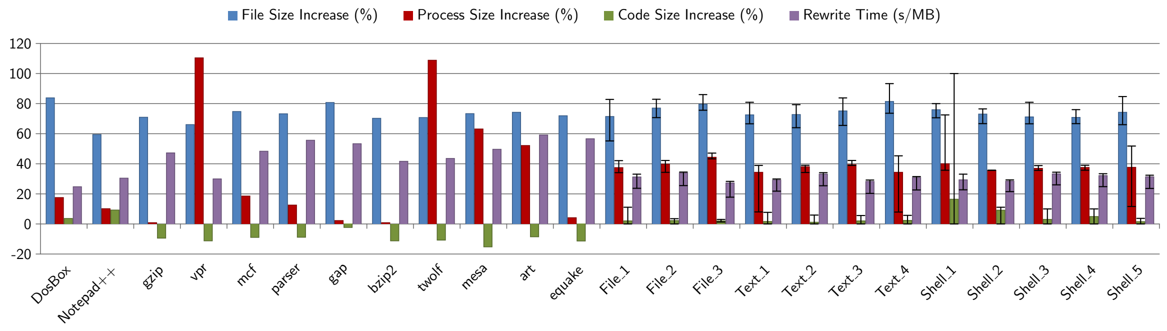


Figure 3.7. Static rewriting times and size increases

Table 3.6. Linux test programs grouped by type and size

Group	**Sizes** (KB)	**Programs**
File_1	17–37	dircolors, ln, mkdir, mkfifo, mknod, mktemp, rmdir, sync
File_2	41–45	chgrp, chmod, chown, dd, rm, shred, touch, truncate
File_3	49–97	chcon, cp, df, dir, install, ls, mv, vdir
Text_1	21–25	base64, cksum, comm, expand, fmt, fold, paste, unexpand
Text_2	25–29	cut, join, md5sum, nl, sha1sum, shuf, tac, tsort
Text_3	29–37	cat, csplit, head, sha224sum, sum, tr, uniq, wc
Text_4	37–89	od, pr, ptx, sha256sum, sha384sum, sha512sum, sort, split, tail
Shell_1	5–17	basename, dirname, env, false, hostid, link, logname, uptime
Shell_2	17–21	arch, echo, printenv, true, tty, unlink, whoami, yes
Shell_3	21	group, id, nice, noshup, pathchk, pwd, runcon, sleep
Shell_4	21–29	chroot, expr, factor, pinky, readlink, tee, test, uname, users
Shell_5	30–85	date, du, printf, seq, stat, stty, su, timeout, who

Figure 3.7 shows how the rewriting phase affects the file size and code section sizes of each binary, which increase on average by 73% and 3% respectively. However, runtime process sizes increase by only 37% on average, with the majority of the increase due to the additional library that is loaded into memory. Our current helper library implementation makes no attempt to conserve its virtual memory allocations, so we believe that process sizes can be further reduced in future development. Occasionally our disassembler is able to safely exclude large sections of static data from rewritten code sections, leading to decreased code sizes. For example, `mesa`'s code section decreases by more than 15%. On average, static rewriting of Windows binaries requires 45 seconds per megabyte of code sections, whereas Linux binaries require 31 seconds per megabyte.

Linux filenames in Fig. 3.7 are grouped by type (File, Text, and Shell) and by program size due to the large number of programs. Table 3.6 lists the programs in each group.

Gadget Elimination

One means of evaluating ROP attack protection is to count the number of gadgets that remain after securing each binary. There are several tools available for such evaluation, including Mona [Corelan Team, 2012] on Windows and RoPGadget [Salwan, 2012] on Linux. We used Mona to evaluate the stirred Windows SPEC2000 benchmark programs. Mona reports the number of gadgets the binary contains after the load-time phase is complete. We define a gadget as *unusable* if it is no longer at the same virtual address after basic block randomization. Figure 3.8 shows that on average STIR causes 99.99% of gadgets to become unusable. The only gadgets that remain after randomization of the test programs consist of a `pop` and a `retn` instruction that happened to fall onto the same address. Most malware payloads are not expressible with such primitive gadgets to our knowledge.

We also applied the Q exploit hardening system [Schwartz et al., 2011] to evaluate the effectiveness of our system. Since Q is a purely static gadget detection and attack payload

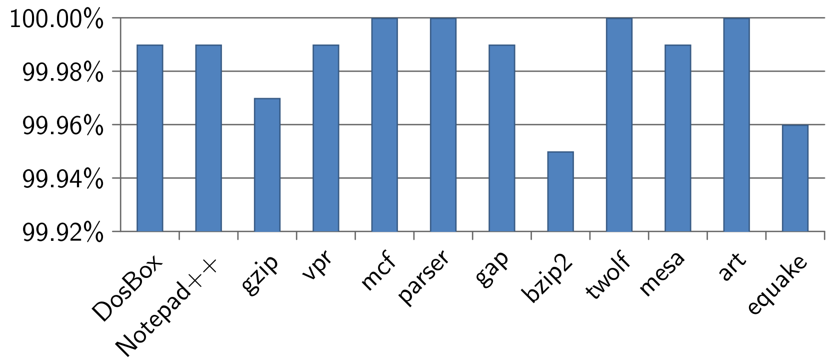


Figure 3.8. Gadget reduction for Windows binaries

generation tool, running Q dynamically after a binary has been stirred is not possible. Instead, we ran Q on a number of Linux binaries (*viz.*, rsync, opendchub, gv, and proftpd) to generate a payload, and then ran a script that began execution of the stirred binary, testing each of the gadgets Q selected for its payload after randomization. Attacks whose gadgets all remained usable after stirring were deemed successful; otherwise, Q's payload fails. In our experiments, no payload generated by Q was able to succeed against STIR.

Entropy

To estimate the entropy that STIR provides we consider a brute-force ROP attack against a web server with an identified vulnerability running on a 32-bit PaX enabled Linux OS. The vulnerability allows an attacker to perform a ROP attack that uses g gadgets. We assume that failed ROP attacks cause the web server to crash and be replaced with a new forked copy with the same randomization parameters. We consider the following two scenarios.

Scenario 1–PaX ASLR enabled PaX's ASLR implementation provides 16 bits of entropy for the starting address of the binary. In the absence of any information leaks, this implies that the start address of the binary can be any one of 2^{16} or 65,536 possible locations. However, since ASLR preserves the internal structure of the binary, correctly guessing the start address is sufficient to determine the starting addresses of each of the g required gadgets.

This implies that we there are only 65,536 different sets of gadget addresses to choose amongst, or about 32,768 guesses in the average case, before our brute–force attack succeeds.

Scenario 2–STIR enabled The STIR helper library performs run–time relocation of the new text segment before scrambling its contents. Our implementation allows the new text segment to be located at any one of 2^{19} possible page–aligned addresses, and thus offers a minimum of 19 bits of entropy without considering stirring.

Once the binary has been stirred, then both the starting address as well as the internal structure of the binary are randomized at run-time. This implies that knowing the start address of the binary, or even that of one of the gadgets will not help locate any other gadgets. Choosing the correct set of g addresses thus means choosing the right set out of $2^{19}!/(2^{19} - g)!$ different possibilities.

The effective entropy can be calculated as:

$$log_2(2^{19}!/(2^{19} - g)!) = \sum_{n=2^{19}-g}^{2^{19}} log_2 n \tag{3.1}$$

Considering an exploit that requires 5 gadgets, we get the effective entropy as almost 95 bits. For an exploit with 50 gadgets we get the effective entropy as approximately 950 bits. For a 100 gadget exploit the entropy is almost 1900 bits. As these calculations show, the effective entropy that STIR provides is far in excess of what ASLR provides.

A comparable test on Linux is not possible since RoPGadget currently only supports purely static detection of gadgets. (STIR randomizes the binary at load-time, causing RoPGadget to return incorrect results.) However, since there is no fundamental difference between STIR's randomization approach on Linux and Windows platforms, we expect that a Linux evaluation would yield comparable results to the ones yielded on Windows.

Performance Overhead

Runtime performance statistics for Windows and Linux binaries are shown in Figs. 3.9 and 3.10, respectively, with each bar reflecting the application's median overhead over 20 trials. The median overhead is 4.6% for Windows applications, 0.3% for Linux applications, and 2.4% overall.

To isolate the effects of caching, Fig. 3.10 additionally reports the runtime overhead (discounting startup and initialization time) of *unstirred* Linux binaries, in which the load-time stirring phase was replaced by a loop that touches each original code byte without rewriting it, and then runs this unmodified, original code. This potentially has the effect of pre-fetching some or all of the code into the cache, decreasing some runtimes (although, as the figure shows, in practice the results are not consistent). Stirred binaries exhibit a median overhead of 1.2% over unstirred ones.

Amongst the Windows binaries, the `gap` SPEC2000 benchmark program consistently returns the worst overhead of 35%. This may be due to excessive numbers of callback functions or computed jumps. In contrast, the `parser` benchmark actually increases in speed by 5%. We speculate that this is due to improved locality resulting from separation of static data from the code (at the expense of increased process size). On average, the SPEC2000 benchmarks exhibit an overhead increase of 6.6%.

We do not present any runtime information for DosBox and Notepad++, since both are user-interactive. We did, however, manually confirm that all program features remain functional after transformation, and no performance degradation is observable.

To separate the load-time overhead of the stirring phase from the rest of the runtime overhead, Fig. 3.11 plots the stirring time against the code size. As expected, the graph shows that the increase in load-times is roughly linear with respect to code sizes, requiring 1.37ms of load-time stirring per KB of code on average.

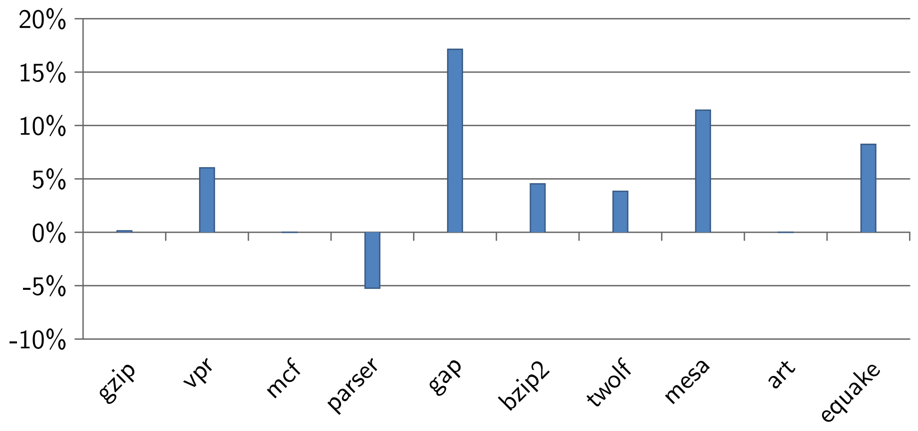


Figure 3.9. Runtime overheads for Windows binaries

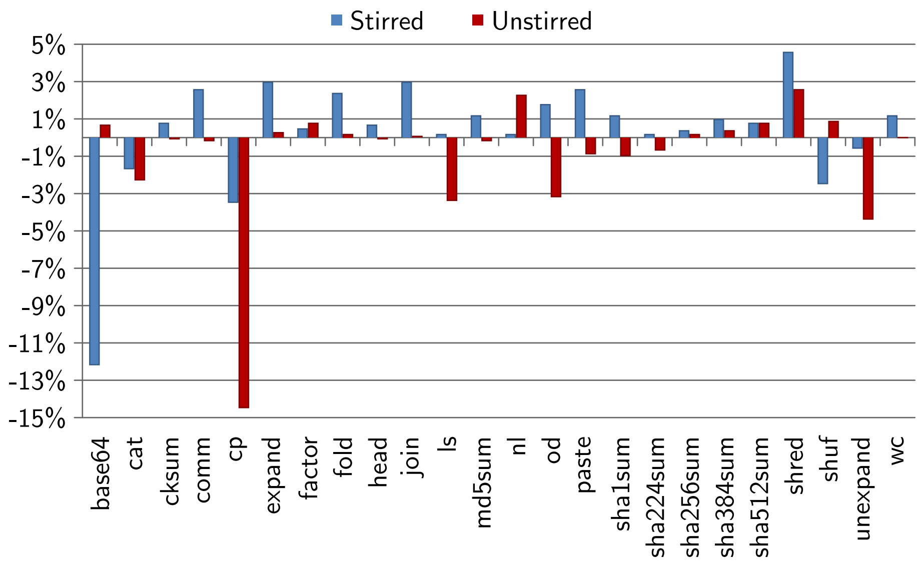


Figure 3.10. Runtime overheads for Linux binaries

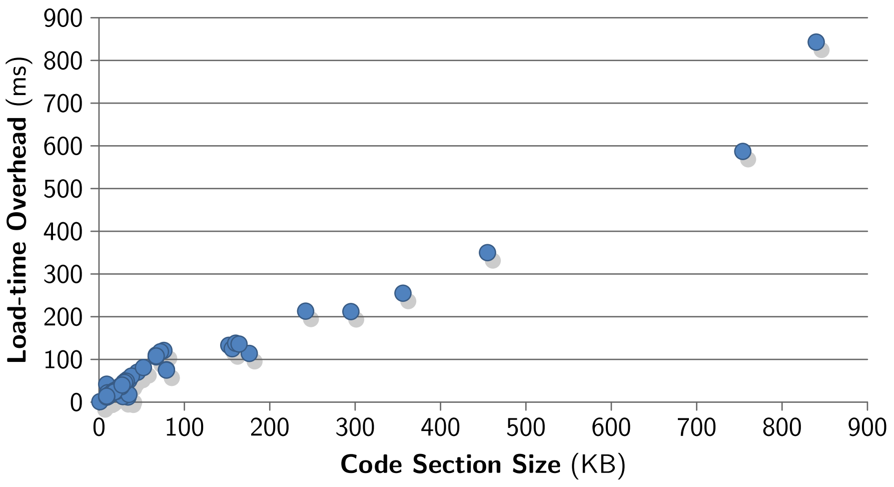


Figure 3.11. Load-time overhead vs. code size

For the most part, none of our tests require manual intervention by the user; all are fully automatic. The only exception to this is that IDA Pro's disassembly of each SPEC2000 benchmark program contained exactly two identical errors due to a known bug in its control-flow analysis heuristic. We manually corrected these two errors in each case before proceeding with static rewriting.

Table 3.7 provides full file info for each binary.

3.3 Reins

Whereas Stir provides probabilistic safety guarantees on the binaries it protects (a brute force attack would be extremely difficult but still possible), a more strictly enforced and machine verifiable form of security is often desired. This section presents the first, purely static, CISC native code rewriting and in-lining system (Reins) using the techniques described in Section 3.1 combined with sandboxing and a machine verifiable verification system.

The rest of this section is structured as follows: An overview of the goals and assumptions used in Reins is outlined in Section 3.3.1. Section 3.3.2 describes the techniques used to constrain the control flow of an x86 binary, including the one used in Reins. The policy specification language is described in Section 3.3.3. Section 3.3.4 details the machine verifiable proof of safety provided with Reins, followed by an evaluation of Reins on real world binaries in Section 3.3.5.

3.3.1 Overview

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 not currently supported. (In fact, rewriting Windows binary code is much more challenging than for Linux due to the much greater diversity of Windows-targeting compilers. Extending Reins to support Linux binaries is merely an engineering task that will be accomplished in the future. Stir supports Linux binaries, supporting this claim.)

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.

Table 3.7. STIR Binary size overheads

	File Sizes (K)			**Code Sizes (K)**			**Process Sizes (K)**			**Rewriting**
Program	**Old**	**New**	**Increase**	**Old**	**New**	**Increase**	**Old**	**New**	**Increase**	**Time**
DOSBox	3640	6701	(+84%)	3015	3133	(+4%)	49.4mb	58.2mb	(+18%)	167.3s
Notepad++	1512	2415	(+60%)	840	920	(+10%)	13400	14800	(+10%)	74.1s
gzip	219	375	(+71%)	176	159	(-10%)	186mb	188mb	(+1%)	17.8s
vpr	386	642	(+66%)	295	261	(-12%)	2.8mb	5.9mb	(+110%)	19.4s
mcf	180	315	(+75%)	152	138	(-9%)	11.2mb	13.3mb	(+19%)	15.3s
parser	294	510	(+73%)	242	220	(-9%)	23.4mb	26.4mb	(+13%)	28.5s
gap	536	970	(+81%)	455	443	(-3%)	198mb	203mb	(+2%)	52s
bzip2	193	329	(+70%)	156	138	(-12%)	184mb	186mb	(+1%)	13.8s
twolf	438	749	(+71%)	356	317	(-11%)	2.2mb	5.7mb	(+159%)	32.8s
mesa	846	1,469	(+74%)	754	637	(-16%)	9.3mb	15.2mb	(+63%)	73.3s
art	192	335	(+74%)	160	146	(-9%)	4.2mb	6.4mb	(+52%)	19.9s
equake	198	341	(+72%)	164	145	(-12%)	45mb	47mb	(+4%)	19.4s
arch	21	35	(+ 64.67%)	10	11	(+ 3.90%)	4272	5796	(35%)	0.25s
base64	25	44	(+ 74.95%)	16	16	(+ 3.85%)	4276	5940	(38%)	0.36s
basename	17	30	(+ 75.19%)	10	10	(+ 4.20%)	4268	5792	(35%)	0.22s
cat	37	68	(+ 81.65%)	27	28	(+ 1.84%)	4288	6096	(42%)	0.58s
chcon	49	87	(+ 76.65%)	34	35	(+ 2.37%)	4444	6368	(43%)	0.80s
chgrp	45	80	(+ 76.54%)	31	32	(+ 2.99%)	4296	6108	(42%)	0.69s
chmod	41	74	(+ 79.15%)	29	30	(+ 3.61%)	4292	6100	(42%)	0.66s
chown	45	81	(+ 79.14%)	32	33	(+ 2.77%)	4296	6108	(42%)	0.73s
chroot	25	42	(+ 65.81%)	13	14	(+ 3.07%)	4276	5936	(38%)	0.31s
cksum	21	35	(+ 66.69%)	11	11	(+ 4.60%)	4272	5796	(35%)	0.25s
comm	25	42	(+ 65.81%)	13	14	(+ 2.57%)	4272	5932	(38%)	0.30s
cp	97	177	(+ 81.45%)	76	77	(+ 0.99%)	4700	6916	(47%)	1.61s
csplit	37	66	(+ 76.44%)	25	26	(+ 3.03%)	4288	6092	(42%)	0.55s
cut	29	52	(+ 78.98%)	19	20	(+ 4.80%)	4280	5948	(38%)	0.45s
date	49	85	(+ 72.70%)	32	33	(+ 2.63%)	4448	6112	(37%)	0.66s
dd	45	79	(+ 73.41%)	30	31	(+ 2.72%)	4444	6104	(37%)	0.66s
df	57	106	(+ 84.14%)	45	46	(+ 1.55%)	4308	6264	(45%)	0.97s
dir	94	165	(+ 76.02%)	67	69	(+ 2.72%)	4696	6772	(44%)	1.48s
dircolors	29	45	(+ 54.52%)	13	13	(+ 2.86%)	4276	5936	(38%)	0.28s
dirname	17	30	(+ 75.06%)	10	10	(+ 4.54%)	4268	5792	(35%)	0.23s
du	85	157	(+ 83.69%)	67	69	(+ 2.22%)	4336	6580	(51%)	1.44s
echo	21	35	(+ 64.35%)	10	11	(+ 4.81%)	4272	5796	(35%)	0.23s
env	17	30	(+ 75.88%)	10	10	(+ 4.26%)	4268	5792	(35%)	0.23s
expand	21	37	(+ 71.81%)	12	12	(+ 3.40%)	4272	5932	(38%)	0.28s
expr	29	51	(+ 74.69%)	18	19	(+ 3.62%)	4280	5944	(38%)	0.42s
factor	25	43	(+ 68.74%)	14	15	(+ 3.96%)	4276	5936	(38%)	0.33s
false	17	29	(+ 70.45%)	9	9	(+ 5.18%)	4268	5792	(35%)	0.20s
fmt	25	43	(+ 71.58%)	14	15	(+ 4.68%)	4308	4652	(8%)	0.33s
fold	21	36	(+ 70.86%)	12	12	(+ 4.02%)	4272	5932	(38%)	0.28s
groups	21	35	(+ 66.02%)	11	11	(+ 3.66%)	4272	5796	(35%)	0.23s
head	33	57	(+ 72.27%)	21	21	(+ 3.53%)	4284	5952	(38%)	0.48s
hostid	17	30	(+ 72.40%)	9	10	(+ 4.91%)	4268	5792	(35%)	0.22s
id	21	37	(+ 75.18%)	13	13	(+ 3.23%)	4416	6056	(37%)	0.30s
install	89	161	(+ 79.72%)	68	69	(+ 1.40%)	4700	6776	(44%)	2.31s
join	29	51	(+ 75.79%)	19	19	(+ 3.25%)	4280	5944	(38%)	0.42s
link	17	30	(+ 73.99%)	9	10	(+ 4.57%)	4268	5792	(35%)	0.22s
ln	37	64	(+ 72.26%)	24	24	(+ 3.03%)	4288	6092	(42%)	0.53s
logname	17	30	(+ 72.96%)	9	10	(+ 4.82%)	4268	5792	(35%)	0.22s
ls	94	165	(+ 76.02%)	67	69	(+ 2.72%)	4696	6772	(44%)	1.47s
md5sum	29	49	(+ 68.74%)	17	17	(+ 2.06%)	4280	5944	(38%)	0.38s
mkdir	37	66	(+ 78.00%)	26	26	(+ 2.90%)	4432	6216	(40%)	0.55s
mkfifo	21	35	(+ 67.19%)	11	11	(+ 4.16%)	4416	5920	(34%)	0.27s
mknod	25	43	(+ 69.17%)	14	15	(+ 4.40%)	4420	6060	(37%)	0.33s
mktemp	29	49	(+ 68.73%)	17	17	(+ 1.53%)	4428	5944	(34%)	0.41s
mv	89	165	(+ 84.16%)	72	73	(+ 1.24%)	4692	6772	(44%)	1.50s

Table 3.7 continued

Program	File Sizes (K) Old	New	Increase	Code Sizes (K) Old	New	Increase	Process Sizes (K) Old	New	Increase	Rewriting Time
`nice`	21	36	(+ 69.16%)	11	12	(+ 3.99%)	4272	5932	(38%)	0.27s
`nl`	29	50	(+ 69.64%)	17	18	(+ 3.22%)	4280	5944	(38%)	0.38s
`nohup`	21	36	(+ 70.57%)	12	12	(+ 2.96%)	4272	5932	(38%)	0.28s
`od`	53	92	(+ 72.88%)	35	36	(+ 2.74%)	4304	6120	(42%)	0.75s
`paste`	21	36	(+ 71.37%)	12	12	(+ 3.48%)	4272	5932	(38%)	0.28s
`pathchk`	21	35	(+ 65.33%)	11	11	(+ 4.22%)	4272	5796	(35%)	0.25s
`pinky`	25	43	(+ 70.74%)	15	15	(+ 2.06%)	4276	5936	(38%)	0.34s
`pr`	49	87	(+ 76.45%)	34	35	(+ 3.82%)	4448	4796	(8%)	0.73s
`printenv`	17	30	(+ 73.15%)	9	10	(+ 5.15%)	4268	5792	(35%)	0.22s
`printf`	41	73	(+ 76.20%)	28	29	(+ 2.84%)	4292	6100	(42%)	0.59s
`ptx`	53	95	(+ 78.66%)	38	39	(+ 2.24%)	4304	6252	(45%)	0.80s
`pwd`	21	38	(+ 77.12%)	13	14	(+ 2.49%)	4272	5932	(38%)	0.30s
`readlink`	29	51	(+ 75.46%)	19	19	(+ 4.25%)	4280	5944	(38%)	0.44s
`rm`	45	80	(+ 77.05%)	31	32	(+ 2.76%)	4296	6108	(42%)	0.70s
`rmdir`	29	53	(+ 82.44%)	21	21	(+ 3.46%)	4280	5948	(38%)	0.45s
`runcon`	21	36	(+ 69.58%)	12	12	(+ 2.65%)	4416	6056	(37%)	0.28s
`seq`	33	59	(+ 78.74%)	23	23	(+ 2.93%)	4284	5952	(38%)	0.48s
`sha1sum`	29	52	(+ 78.93%)	20	20	(+ 1.82%)	4280	5948	(38%)	0.45s
`sha224sum`	37	66	(+ 77.56%)	26	26	(+ 1.33%)	4288	5960	(38%)	0.58s
`sha256sum`	37	66	(+ 77.56%)	26	26	(+ 1.33%)	4288	5960	(38%)	0.58s
`sha384sum`	89	172	(+ 92.51%)	79	80	(+ 0.50%)	4340	6068	(39%)	1.41s
`sha512sum`	89	172	(+ 92.51%)	79	80	(+ 0.50%)	4340	6068	(39%)	1.41s
`shred`	45	78	(+ 72.30%)	30	30	(+ 1.36%)	4444	6104	(37%)	0.64s
`shuf`	29	51	(+ 74.03%)	19	19	(+ 2.07%)	4428	5944	(34%)	0.42s
`sleep`	21	35	(+ 63.67%)	10	11	(+ 4.55%)	4268	5792	(35%)	0.25s
`sort`	73	130	(+ 76.42%)	52	54	(+ 3.15%)	4468	6408	(43%)	1.16s
`split`	45	81	(+ 78.87%)	32	33	(+ 1.94%)	4296	6108	(42%)	0.72s
`stat`	41	69	(+ 67.09%)	25	25	(+ 1.11%)	4436	6088	(37%)	0.53s
`stty`	53	88	(+ 65.33%)	31	32	(+ 3.02%)	4304	4800	(12%)	0.67s
`su`	30	50	(+ 64.97%)	18	18	(+ 0.40%)	4388	6032	(37%)	0.36s
`sum`	29	51	(+ 72.73%)	18	19	(+ 3.46%)	4280	5944	(38%)	0.41s
`sync`	17	30	(+ 72.08%)	9	10	(+ 5.02%)	4268	5792	(35%)	0.22s
`tac`	25	42	(+ 65.89%)	14	14	(+ 2.40%)	4284	5944	(38%)	0.31s
`tail`	49	89	(+ 79.42%)	35	37	(+ 3.88%)	4300	4800	(12%)	0.77s
`tee`	21	35	(+ 66.22%)	11	11	(+ 4.12%)	4272	5796	(35%)	0.25s
`test`	25	44	(+ 72.48%)	15	16	(+ 3.45%)	4268	5932	(38%)	0.34s
`timeout`	38	68	(+ 80.13%)	28	28	(+ 1.51%)	4288	6096	(42%)	0.59s
`touch`	41	70	(+ 70.13%)	26	26	(+ 2.21%)	4440	5964	(34%)	0.59s
`tr`	33	57	(+ 71.45%)	20	21	(+ 4.33%)	4292	5960	(38%)	0.44s
`true`	17	29	(+ 70.45%)	9	9	(+ 5.18%)	4268	5792	(35%)	0.20s
`truncate`	41	75	(+ 81.62%)	30	31	(+ 1.81%)	4292	6100	(42%)	0.64s
`tsort`	25	41	(+ 64.00%)	13	13	(+ 3.57%)	4276	5936	(38%)	0.30s
`tty`	17	30	(+ 72.27%)	9	10	(+ 4.41%)	4268	5792	(35%)	0.22s
`uname`	21	35	(+ 64.67%)	10	11	(+ 3.90%)	4272	5796	(35%)	0.23s
`unexpand`	21	38	(+ 76.45%)	13	13	(+ 3.25%)	4272	5932	(38%)	0.30s
`uniq`	29	48	(+ 65.82%)	16	16	(+ 2.45%)	4276	5940	(38%)	0.36s
`unlink`	17	30	(+ 73.51%)	9	10	(+ 4.55%)	4268	5792	(35%)	0.22s
`uptime`	5	9	(+ 75.97%)	1	2	(+ 90.97%)	2084	3596	(72%)	0.03s
`users`	21	35	(+ 63.91%)	10	11	(+ 4.26%)	4268	5792	(35%)	0.25s
`vdir`	94	165	(+ 76.01%)	67	69	(+ 2.72%)	4696	6772	(44%)	1.49s
`wc`	29	51	(+ 72.41%)	18	18	(+ 2.04%)	4280	5944	(38%)	0.41s
`who`	41	72	(+ 73.94%)	27	28	(+ 1.88%)	4288	6096	(42%)	0.58s
`whoami`	17	30	(+ 73.11%)	9	10	(+ 4.64%)	4268	5792	(35%)	0.22s
`yes`	17	30	(+ 72.99%)	9	10	(+ 5.05%)	4268	5792	(35%)	0.22s
median			(+73.3%)			(+2.8%)			(+37.5%)	5.3s

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. This approach to security is known as *sandboxing*.

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.

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. Since we are already placing guards around all computed jumps, as outlined in Section 3.1, any added logic can be added to those guard instructions to ensure proper control flow, as explained in Section 3.3.2.

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 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 [Erdélyi, 2008] 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. 3.12. 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.

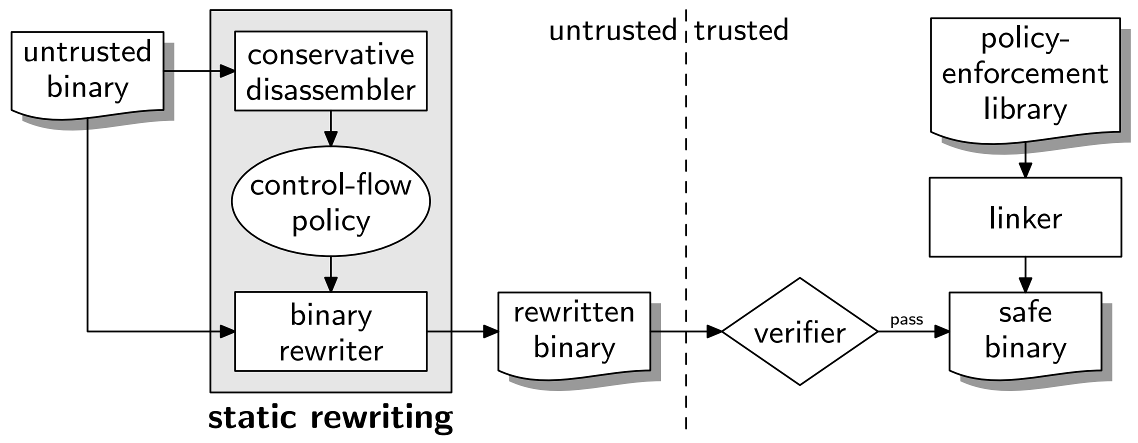


Figure 3.12. Reins architecture

3.3.2 Control-flow Safety

Our binary rewriting algorithm uses Software Fault Isolation (SFI) [Wahbe et al., 1993] to constrain control-flows of untrusted code. It is based on an SFI approach pioneered by PittSFIeld [McCamant and Morrisett, 2006], 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 low-order and high-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. 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. The details of the lookup table were described in Section 3.1.

Table 3.8 shows the transformations used in Reins and how they differ from the original transformation set shown in Table 3.4. When the original computed jump employs a memory operand instead of a register, as shown in row 2 of Table 3.8, the rewritten code requires a scratch register. Table 3.8 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 [Fog, 2009].[3]

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 3.8, 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:

[3]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 required this, so our experiments use the table's shorter sequence.

Table 3.8. 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`]`

Callback registrations	`call/jmp [IAT:`n`]`	`call/jmp tramp_`n
		`tramp_`n`:`
		    `push` ⟨*registration function address*⟩
		    `call intermediary.reg_callback`
		`return_tramp:`
		    `call intermediary.callback_ret`

Dynamic linking	`call [IAT:GPA]`	`push offset tramp_pool`
		`call [IAT:GPA]`
		`tramp_pool:`
		    `.ALIGN` c
		    `call intermediary.dll_out`
		    `.ALIGN` c
		    `call intermediary.dll_out`
		    ⋮

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 3.8. 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.

This functionality along with dynamic linking and callbacks have already been described in more detail in Section 3.1, as well as in the technical report [Hamlen et al., 2010].

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.

Reins Instrumentation Examples. The instrumentation examples in Figures 3.1 and 3.2 have been updated according to the specifications of Reins in Figures 3.13 and 3.14.

Figure 3.13 implements a register-indirect call to a system API function (MBTWC). This differs from Figure 3.1 because a mask is introduced in front of the `call` instruction to support the Reins security policy. Since this is the case, the `mov` instruction has to point to trampoline code, which performs a safe jump to the same destination. This ensures that the

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 3.13. REINS exmaple of a register-indirect system call

mask instruction does not ruin the imported function address while making the call provably safe to execute.

Figure 3.14 needs less modification than Figure 3.13. The addition of the mask instruction and chunk alignment satisfy the requirements of REINS, allowing for safe execution.

3.3.3 Policy Specification

To quickly and easily demonstrate the REINS' 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).

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 3.14. Reins example using a jump table

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

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

policy  = e1* + e2*;
```

Figure 3.15. 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 3.16. Eureka email policy

To illustrate, Fig. 3.15 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, `*` 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. This collection was inspired by the work in [Jones and Hamlen, 2010] that has identified these as sufficient to enforce an array of interesting, practical policies with other IRM systems. More extensive collections of dynamic tests are important for supporting more expressive policy languages, but is reserved as a subject for future work.

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 policy as a determinized *security automaton* [Alpern and Schneider, 1986]—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.

3.3.4 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:

1. All executable sections reside in low memory.
2. All exported symbols (including the program entrypoint) target low memory chunk boundaries.
3. No disassembled instruction spans a chunk boundary.
4. Static branches target low memory chunk boundaries.
5. All computed jump instructions that do not reference the IAT are immediately preceded by the appropriate `and`-masking instruction from Table 3.8 in the same chunk.
6. 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.)
7. 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.

3.3.5 Evaluation

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 §3.4.3. 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 §3.3.2. 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 §3.3.4. 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.

Rewriting Effectiveness

We tested Reins with a set of binary programs listed in Tables 3.9 and 3.10. Table 3.9 lists results for some of the benchmarks from the SPEC 2000 benchmark suite. Table 3.10

Table 3.9. Experimental results: SPEC benchmarks

Binary Program	**Size Increase**			**Rewriting Time** (s)	**Verification Time** (ms)
	File (%)	**Code** (%)	**Process** (%)		
`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

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.

The applications in Tables 3.10 and 3.9 were chosen to be reasonably large and realistic yet relatively self-contained. Each includes most or all of the difficult Windows binary features

Table 3.10. Experimental results: Applications and malware

Binary Program	**Size Increase**			**Rewriting Time** (s)	**Verification Time** (ms)
	File (%)	**Code** (%)	**Process** (%)		
`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

discussed in Chapter 1, but statically or dynamically links to not more than about five standard system libraries and local modules. This helped keep the engineering task reasonable for research purpose. Supporting less self-contained applications requires manually specifying a trusted interface for each new trusted system library, which can become a significant task when the number of trusted libraries is large. For example, Microsoft Office consists of over 100 different local modules that statically link to at least 50 different system libraries, and that dynamically link to innumerable other modules via COM services. Supporting such sprawling applications essentially requires huge amount of engineering effort to supporting the entire Windows runtime, which is not something we are prepared to undertake for research purpose. Therefore, Tables 3.10 and 3.9 are limited to mid-size applications that are nonetheless too complex to be supported by any past SFI/IRM system without access to source code or debugging information.

Performance Overhead

We also measured the performance of the non-interactive programs in Tables 3.9 and 3.10. The runtimes of the rewritten programs as a percentage of the runtimes of the originals is presented in Fig. 3.17. The median overhead is 2.4%, and the maximum is approximately 15%. As with other similar works [Abadi et al., 2009, Ford and Cox, 2008], 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 3.9 and 3.10 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 §3.3.6.

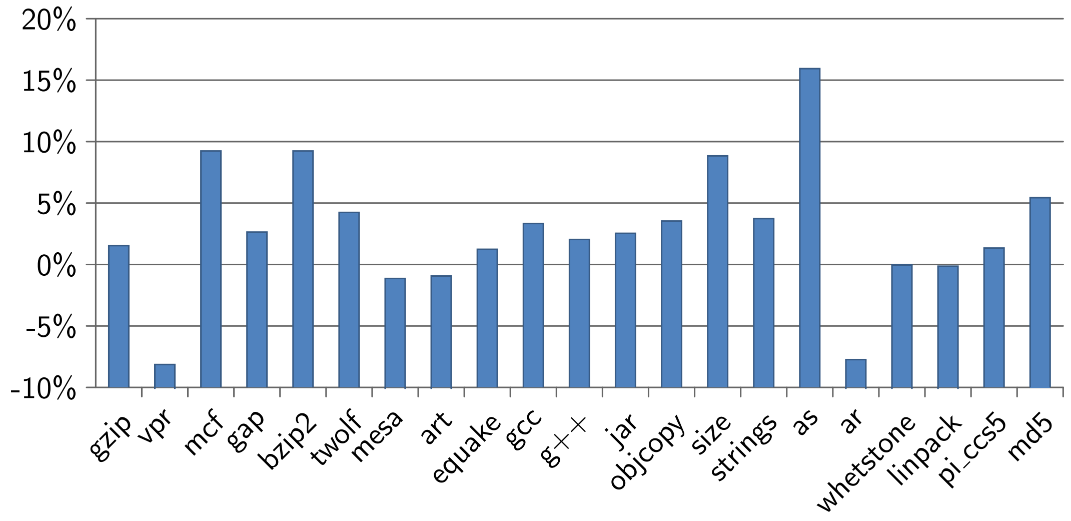


Figure 3.17. Runtimes of rewritten binaries relative to originals

3.3.6 Case Studies

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 §3.3.2. 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. 3.16, 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. 3.15, 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.

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 pre-compiled 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 3.10. 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.

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. 3.16 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.

3.4 Intermediary Library

Section 3.1.4 explains that in order to preserve the semantics of policy-permitted system calls, the rewriting algorithm must provide a safe mechanism to exit the sandbox, and this mechanism must accommodate the myriad system call forms exhibited by real-world COTS binaries. In addition, dynamically linked system calls are calculated at runtime and therefore we implement a mechanism to support dynamic interception or some good control flows to trusted libraries will not be preserved as they will be mistakenly treated as untrusted. And finally, since we are mediating all system calls, any security policy that we enforce on system calls must also mediate dynamic library calls. All of these tasks are accomplished by the intermediary library, whose implementation details are detailed in this section.

3.4.1 Callback Handling

Section 3.1.4 outlines the methods used to handle callback instructions, but omits some details of callback handling. These details are covered in this section. Figure 3.18 shows a normal execution of a call to `msvcrt::atexit`, which is a `libc` callback registration function whose single code pointer argument is later called by the system at process exit.

The execution of an `atexit` callback proceeds as follow:

1. A callback address (the location of a program termination cleanup method within the binary) is pushed on to the stack as a parameter to `atexit` and the call to `atexit` is executed.

2. The callback address is stored for later use and the execution of `atexit` proceeds until it returns.

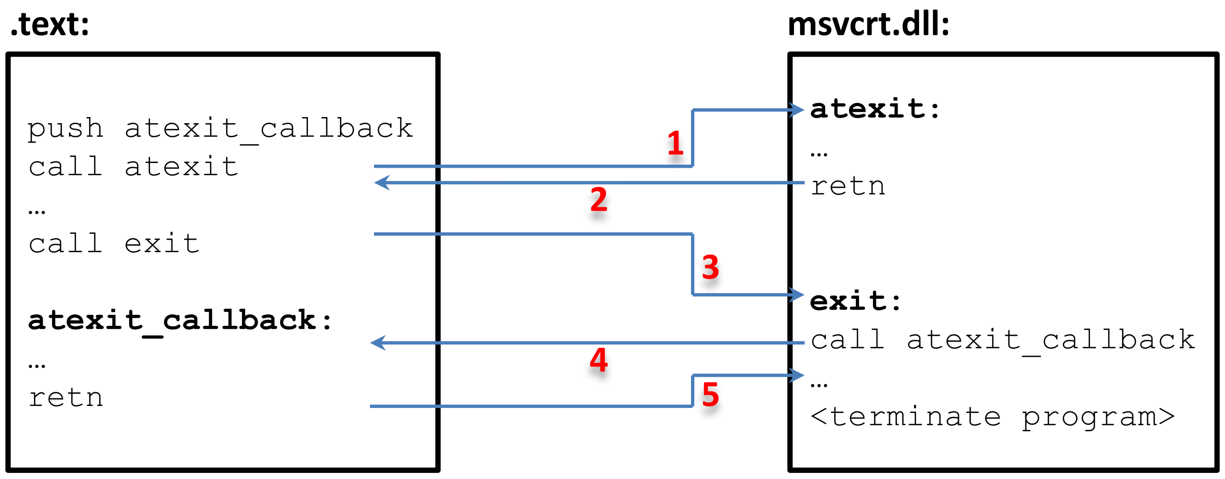


Figure 3.18. Example of `.atexit` callback

3. The binary continues executing, and when it needs to terminate, it calls `msvcrt::exit`.

4. `exit` calls the previously stored callback address, executing the binary's termination cleanup method.

5. The callback method execution finishes, returns to `exit`, which finishes executing and terminates the binary.

This is a rather simple example of a callback. Both the callback storage strategy and the circumstances of callback invocation vary from callback to callback. For example, `initterm` receives a length-2 array of callback pointers instead of a single callback pointer. CreateWindow doesn't push a callback address onto the stack but rather a pointer to a standard structure with a callback address within it. That structure has to be parsed in order for this type of callback to be supported.

Figure 3.19 displays the control flow of a rewritten program that executes a call to `msvcrt::atexit`. The first step in supporting callbacks is recognizing all of them during the rewriting process and replacing each branch to a callback instruction with a branch

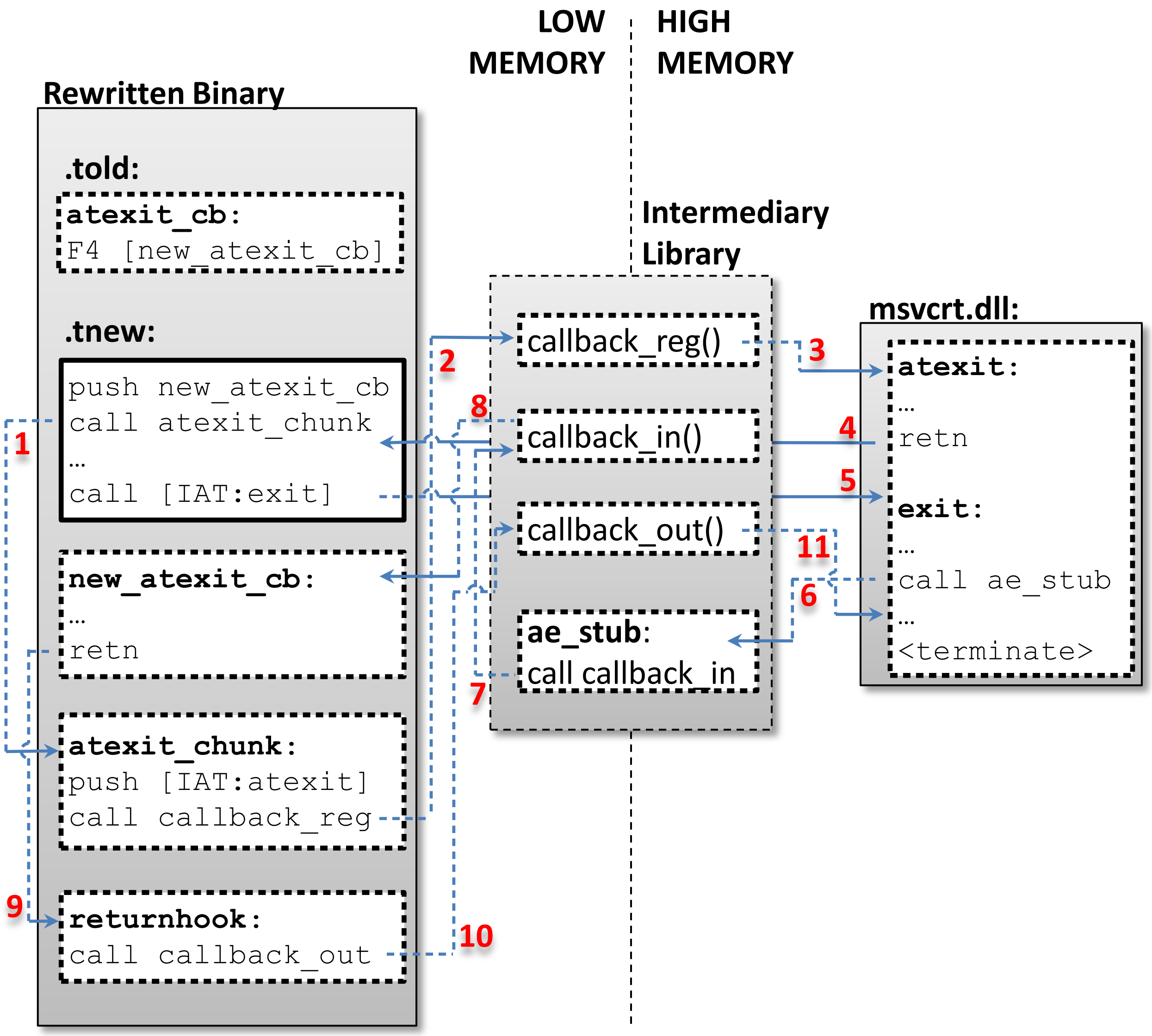


Figure 3.19. Rewritten `.atexit` callback

Table 3.11. Callback Trampoline Chunk Redirection

Original code	**Rewritten code**
`call [IAT:atexit]`	`call offset trampoline_atexit`
	`trampoline_atexit:`
	`   push [IAT:atexit]`
	`   call [IAT:callback_reg]`

to a callback trampoline chunk at the end of the rewritten binary. Table 3.11 shows this transformation for the call to `atexit`.

With the transformations described in Table 3.11, the redirected `atexit` callback proceeds as follows:

1. The program executes until it reaches the call to `atexit` which has been replaced with a call to `atexit_chunk`.

2. `atexit_chunk` pushes the address of `atexit` onto the stack and then calls a callback registration function, `callback_reg`.

3. The callback registration function finds the argument on the stack that is a callback address, replaces it with the address of a trampoline within the intermediary library that will only be used by `atexit`, pops the address of `atexit` off the stack and passes execution to it.

4. `atexit` executes until it returns, passing execution back to the original binary.

5. The binary continues executing, and when it needs to terminate, it calls `msvcrt::exit`.

6. The `exit` method proceeds until it needs to call the callback function, and passes execution to the trampoline in the intermediary library.

7. The trampoline passes execution to `callback_in`.

8. `callback_in` replaces the return address on the stack with the address of `returnhook`, creates a new fiber (in the case of the Windows implementation) for the callback code, since the untrusted callback code could potentially corrupt the stack of the trusted system call. `callback_in` executes the new fiber.

9. Execution of the callback procedure takes place in the new fiber until it returns, at which point the return branches to `returnhook`.

10. `returnhook` passes execution to `callback_out` through the IAT.

11. `callback_out` returns execution to the parent fiber, deletes the callback fiber, and passes execution back to `exit`, which terminates the binary.

Algorithm 5: Callback Registration

```
// Stores known callback function signatures
trusted_cbs [] ← load_signatures() ;
callback_reg(regfunc) begin
    id ← regfunc_id(regfunc) ;
    if trusted_cbs[id].type = PTR_TO_CALLBACK then
        fix_callback(trusted_cbs[id].callback_location) ;
    else if trusted_cbs[id].type = PTR_TO_CALLBACK_ARRAY then
        forall callback in trusted_cbs[id].callback_location do
            fix_callback(callback) ;
    else if trusted_cbs[id].type = PTR_TO_STRUCT then
        forall callback in parse_struct(trusted_cbs[id].callback_location) do
            fix_callback(callback) ;
    else
        fail("Unknown Callback") ;
    jump (regfunc) ;
end
```

Algorithm 5 shows pseudocode for callback_reg(). The main functionality of callback_reg is to determine the function signature of the callback, find the location of the callback address or addresses on the stack, and redirect them using the function fix_callback. The intermediary library generates a static number of callback *stubs* (located within the intermediary library) when it is loaded that fix_callback uses for redirection. Each time fix_callback is called to redirect a callback address, it stores the address of the new callback code located in `.tnew` in one of the stubs, and then replaces the callback on the stack with the address of callback_in. Finally, callback_reg calls the trusted system call stored in regfunc.

It is worth mentioning that this method of handling callbacks is not necessary for all rewriters. This callback handling methodology is necessary for Reins, but Stir does not have a separation of memory or use masks, and therefore doesn't need to use callback stubs, callback_in or callback_out to preserve functionality since the system call can directly branch to the callback rather than going through the intermediary library. Instead, callback_reg can directly replace the callback addresses on the stack for Stir. However, since the library is trusted and the binary is not, this method of system call interception preserves only good control flows, and and allows our rewriting framework to support the myriad callbacks that virtually all Windows binaries require for correct execution.

3.4.2 Dynamic Library Loading

Libraries and functions can be loaded at runtime using different dynamic approaches, such as calling `LoadLibrary`, `GetProcAddress`, or implementing code that executes the functionality of `GetProcAddress` at runtime. For a rewriter like Stir, this does not cause any problem since the address of a loaded function is stored in a register, called, or jumped to at runtime, the system call executes and returns. Figure 3.20 shows the execution of a call to `GetProcAddress` in a normal binary.

When `GetProcAddress` executes, the module and function name of the required system call are passed as parameters, and the address is returned in `eax`. In most cases, that

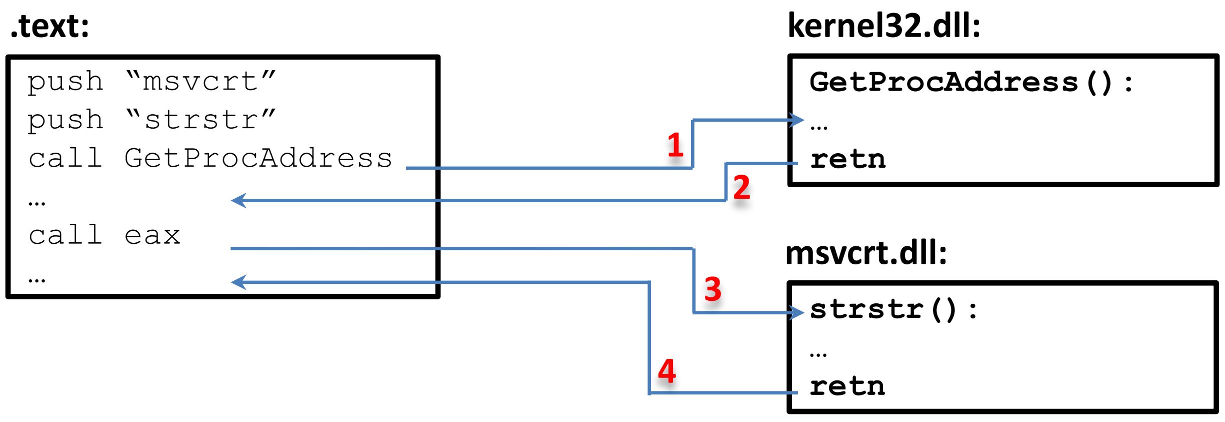


Figure 3.20. Example of GetProcAddress() Execution

instruction is immediately followed by `call eax`. In REINS, the instruction `call eax` is preceded by a mask instruction which will break the execution of the program if `eax` holds a high memory (i.e., trusted) address at this point. Therefore, some mechanism is needed that can target dynamically loaded library functions through low-memory trampolines.

It should also be noted that when this occurs in STIR, if the function pointer that `GetProcAddress` returns contains a callback, it will not be caught by any of the instrumentation we have discussed. Any solution for dynamic loading must solve this problem as well.

To handle this, the rewriter needs to redirect the call to `GetProcAddress` to a wrapper function that returns an appropriate address that is already a low-memory chunk boundary so that it won't be sanitized to a different address by the mask instruction. Figure 3.21 shows the control flow of the program after rewriting, with proper handling of `GetProcAddress`.

The first step for handling `GetProcAddress` is to have addresses ready for `GetProcAddress` to return that are within low memory. This is accomplished by estimating the number of unique calls to `GetProcAddress` within the binary and attaching a trampoline chunk to

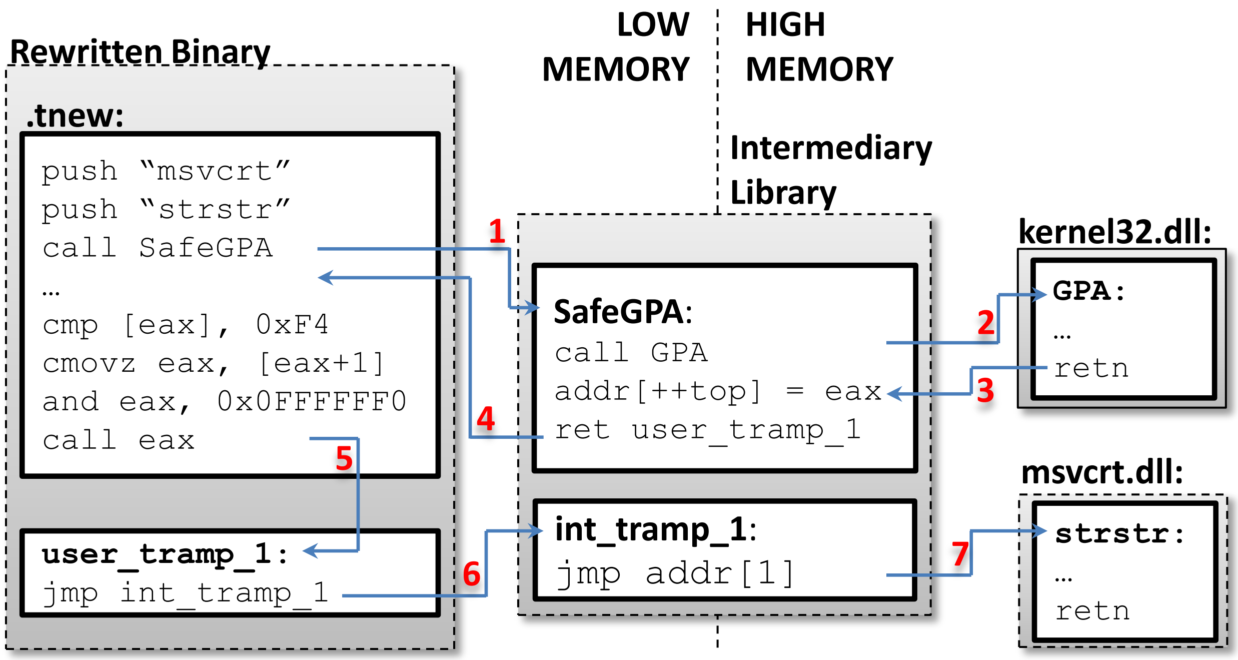


Figure 3.21. Rewritten GetProcAddress() Execution

the end of the binary for each one. Each of these trampolines contains a direct jump to a trampoline within the intermediary library. We then redirect all branches to `GetProcAddress` to a trusted wrapper function, SafeGetProcAddress, in our intermediary library. This wrapper function stores the library function address in the intermediary library trampoline, and then returns the address of the associated user code trampoline.

When the `call eax` instruction occurs and gets masked, the mask doesn't modify any bits since the trampoline is within low memory, and the binary still passes verification according to our security policy.

Two outlier cases must also be handled in order to support all binaries. First, if `GetProcAddress` is called with `GetProcAddress` as the argument, `SafeGetProcAddress` must return a trampoline address that points to `SafeGetProcAddress` rather than to `GetProcAddress`; otherwise when it is used, the resulting call to `GetProcAddress` would not use a trampoline and execution would halt. This may seem like a trivial outlier since no program should need to ask `GetProcAddress` to locate `GetProcAddress` itself. Nevertheless, many COTS binaries we studied exhibit this peculiar behavior.

A few binaries we studied recreate the semantics of `GetProcAddress` within the binary code, and therefore miss the measures we have put in place to facilitate dynamic loading. To handle this problem, we currently manually detect the issue and modify the instruction sequence to instead use `SafeGetProcAddress`. Future work should consider automating some of these cases through a heuristic static analysis.

3.4.3 Policy Implementation

The intermediary library is responsible for callback registration, callbacks, and dynamically linking, forming a foundation for mediating all security-relevant system functions. Thus, our intermediary library also contains the IRM logic that enforces system call policies. The policy specification language for Reins consists of three components: function definitions, event definitions based on defined functions, and a policy definition of permissible event combinations.

Function definitions are simply function signatures that can be matched to all references made to the system call when the binary is rewritten so that the appropriate callback chunks can be created within the binary, and wrapper functions can be created within the intermediary library.

Event definitions specify which arguments constitute security events. Arguments can be supplied as regular expressions (e.g., "*.exe"), conditionals (e.g., $x < 30$), specific values, or not at all to match all values.

Finally, a policy of permissible event sequences is specified as a regular expression. Using these three components, when the intermediary library is compiled, the policy is used to create a deterministic finite automata (DFA) and assign security states to each state of the DFA. A global security state is then stored in the intermediary library, and updated by every security relevant event.

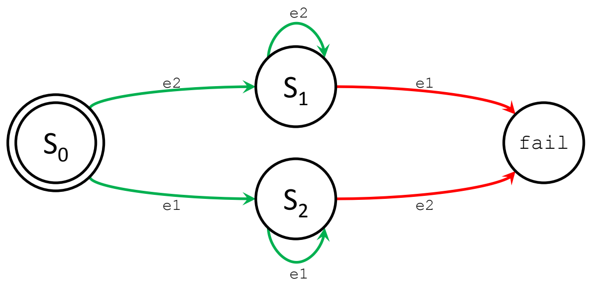


Figure 3.22. Policy DFA Example

To demonstrate the policy synthesis process please refer to Figure 3.15. This policy is first converted into a DFA, which is displayed in Figure 3.22.

The DFA in Figure 3.22 is converted to wrapper functions, which control the security state in the intermediary library. Algorithm 6 shows the wrapper code that is produced to protect the executable.

Algorithm 6: Security Policy Wrapper Functions

```
// Security State
state ← 0 ;
CreateFileW_wrapper(name)  begin
    if name.Contains(".exe") then
        if state = 0 then
            state ← 1 ;
        else if state = 2 then
            fail("Security Policy Violated") ;
    CreateFileW(name) ;
end
connect_wrapper(a, sock)  begin
    if sock = 25 then
        if state = 0 then
            state ← 2 ;
        else if state = 1 then
            fail("Security Policy Violated") ;
    connect(a, sock) ;
end
```

CHAPTER 4

REWRITER TRANSPARENCY

The preceding chapters explicate the design and implementation of a COTS native code rewriting system that provably enforces safety policies, including control-flow and memory safety policies. Although the safety of rewritten programs (i.e., soundness) is formally machine-verified, preservation of the behaviors of policy-adherent programs (i.e., transparency) is not.

This chapter examines the latter challenge in the context of machine-verifying transparency of IRMs implemented in a type-safe bytecode language. While our transparency verification of type-safe IRMs cannot yet be applied to native code IRMs, it is a first step toward such verification.

This chapter presents the design and implementation of the first automated transparency-verifier for IRMs. Our main contributions are as follows:

- We show how prior work on safety-verifiers based on model-checking [Hamlen et al., 2012, Sridhar and Hamlen, 2010] can be extended in a natural way to additionally verify IRM transparency.

- We demonstrate how an untrusted, external invariant generator can be leveraged to reduce the state-exploration burden and afford the verifier a higher degree of generality than the more specialized rewriting systems it checks.

- Prolog unification [Shapiro and Sterling, 1994] and Constraint Logic Programming (CLP) [Jaffar and Maher, 1994] is leveraged to keep the verifier implementation simple and closely tied to the underlying verification algorithm.

- Proofs of correctness are formulated for the underlying verification algorithm using Cousot's abstract interpretation framework [Cousot and Cousot, 1992].
- The feasibility of our technique is demonstrated through experiments on a variety of real-world ActionScript bytecode applets.

Section 4.1 details the ActionScript bytecode language and justifies its use as a stepping stone in binary level IRM transparency verification. The details of how to prove transparency for an IRM system are listed in Section 4.2. The design and theory of our transparency verifier are detailed in Section 4.3. Section 4.4 shows empirical results of our transparency verification algorithm on real world ActionScript ads. Finally, Section 4.5 concludes the chapter with a summary of challenges related to extending this work to native code domains.

4.1 ActionScript Byte Code

ActionScript is a powerful, type-safe mobile code language similar to Java bytecode. Its ubiquity in many modern web browsing technologies, such as Flash ads, makes it a deserving subject of software security research. For example, numerous malware attacks have used ActionScript as a vehicle within the past few years to exploit VM buffer overflow vulnerabilities [Dowd, 2008], perform cross-site-scripting, deny service (e.g., by memory-corruption), and implement click-jack attacks [Mitre Corporation, 2010a,b]. Policies relevant to web ads are often fusions of constraints prescribed by multiple independent parties, including ad distributors and embedding page publishers, who lack access to the applet source code and therefore benefit from the IRM enforcement approach. VM-based security mechanisms for ActionScript are currently limited to enforcing standard coarse-grained policies, such as file system access controls, that are inadequate to reliably distinguish malicious from benign code in many cases.

4.2 IRM Transparency

Past work defines IRM transparency in terms of an abstract relation $\approx : M \times M$ that relates original and IRM-instrumented programs if and only if they are observably, behaviorally equivalent when the original program is policy-adherent [Hamlen et al., 2006]. The abstract relation can be instantiated in terms of various forms of trace equivalence [Ligatti et al., 2005]. Following this approach, we define transparency as follows:

Definition 4.2.1 (Events, Traces, and Policies)**.** *A* trace τ *is a (finite or infinite) sequence of* observable events*, where observable events are a distinguished subset of all* program operations*— instructions parameterized by their arguments. Policies $\mathcal{P}$ denote sets of permissible traces.*

Definition 4.2.2 (Transparency)**.** Programs m *are functions from initial states χ_0 to traces. (Any non-determinism in the program is modeled as an infinite subset of χ_0.) Original program m_1 and its IRM-instrumentation m_2 are* transparent *for policy $\mathcal{P}$ if and only if for every initial state χ_0, $(m_1(\chi_0) \in \mathcal{P}) \Rightarrow (m_1(\chi_0) = m_2(\chi_0))$.*

In our implementation, observable events include most system API calls and their arguments, which are the only means in ActionScript to affect process-external resources like the display or file system. Policy specifications may identify certain API calls as unobservable by assumption, such as those known to be effect-free.

This definition of transparency permits IRMs to augment untrusted code with unobservable stutter-steps (e.g., runtime security checks) and observable interventions (e.g., code that takes corrective action when an impending violation is detected), but not new operations that are observably different even when the original code does not violate the policy. The IRM must also not insert potentially non-terminating loops to policy-adherent flows, since these could suppress desired program behaviors.

Verification

Our transparency verifier is an abstract interpreter and model-checker that non-deterministically explores the cross-product of the state spaces of the original and rewritten programs. To accommodate IRMs that introduce new methods, abstract interpretation is fully interprocedural; calls flow into the bodies of callees. (Recursive and mutually recursive callees require a loop invariant, discussed below, in order for this process to converge.)

Abstract states include an abstract store that maps local variables and object fields to symbolic expressions, abstract traces that describe the language of possible traces exhibited by each program by the time execution reaches each code point, and the various other structures that populate ActionScript VM states. They additionally include linear constraints introduced by conditional instructions. For example, the abstract states of control-flow nodes dominated by the positive branch of a conditional instruction that tests `(x<=y)` contain constraint $x \leq y$.

The verifier attempts to prove that every finite prefix of every flow through the state space has a finite extension at which the traces of the original and rewritten code become equal. It assumes that the original program is policy-adherent, since behavioral changes are permitted (and mandated) for policy-violating flows in original code. Thus, flows in the cross-product space that are reachable only via such violations are pruned from transparency verification.

The cross-product state space is potentially very large even with such pruning, but it is greatly reduced with the aid of an untrusted, external *invariant generator* that hints at how the verifier can converge more quickly on a proof of transparency. For each code point p in the cross-product machine (i.e., each pair of original and rewritten code points), the invariant generator suggests (1) an abstraction of the program states of all flows that reach p, and (2) a post-dominating set of control-flow nodes where flows that pass through p later exhibit equivalent trace prefixes. The former identifies extraneous information inferred by the abstract interpreter that is irrelevant for proving transparency, and that can therefore be safely discarded by the verifier to reduce the search space. The latter is a witness that proves that even if the two traces are not equal at p, they eventually return to equality within a finite number of execution steps. This allows one or both programs to include extra, unobservable operations (stutter-steps) without violating the transparency requirement.

Hints provided by the invariant-generator remain strictly untrusted by the verifier. They are only accepted if they abstract information already inferred by the trusted abstract interpreter. Over-abstractions can cause the verifier to discard information needed to prove transparency, resulting in conservative rejection of the code; but they never result in improper acceptance of non-transparent code. This allows invariant-generation to potentially rely on untrusted information, such as the binary rewriting algorithm, without including that information in the TCB of the system.

To verify abstract states suggested by the untrusted invariant-generator, prune policy-violating flows, and check trace-equality of abstract states, the heart of the transparency verifier employs a model-checking algorithm that proves implications of the form $A \Rightarrow B$, where A is an abstract state inferred by the abstract interpreter, and B is an untrusted abstraction suggested by the invariant-generator. Model-checking consists of two stages:

1. *Unification.* Abstract program states include data structures such as ActionScript bytecode operand stacks, objects, and traces. These are first mined for equality constraints through unification. For example, if abstract program state A includes constraints $\hat{\rho}_1 = v_1{::}\hat{s}_1$, $\hat{\rho}_2 = v_2{::}\hat{s}_2$, and $\hat{\rho}_1 = \hat{\rho}_2$, then unification infers additional equalities $v_1 = v_2$ and $\hat{s}_1 = \hat{s}_2$.

2. *Linear constraint solving.* The equality constraints inferred by step 1 are then combined with any inequality constraints in each abstract state to form a pure linear constraint satisfaction problem without structures. A linear constraint solver is then applied to verify that sentence $A' \wedge \neg B'$ is unsatisfiable, where A' and B' are the linear constraints from A and B, respectively.

Both unification and linear constraint solving can be elegantly realized in Prolog with Constraint Logic Programming (CLP) [Jaffar and Maher, 1994], making this an ideal language for our verifier implementation.

Verification assumes bytecode type-safety of both original and rewritten code as a prerequisite. This assumption is checked by the ActionScript VM type-checker. Assuming type-safety allows the IRM and verifier to leverage properties such as object encapsulation, memory safety, and control-flow safety to reduce the space of attacks that must be anticipated.

Invariant Generation

General-purpose invariant-generation is well known to be intractable for arbitrary software. However, IRM systems typically leave large portions of the untrusted programs they modify unchanged for practical reasons. Their modifications tend to be limited to small, isolated blocks of guard code scattered throughout the modified binary. Past work has observed that the unmodified sections of code tend to obey relatively simple invariants that facilitate tractable proofs of soundness for the resulting IRM [Sridhar and Hamlen, 2010].

We observe that a similar strategy suffices to generate invariants that prove transparency for these IRMs. Specifically, an invariant-generator for a typical IRM system can assert that if the two programs are observably equivalent on entry to each block of guard code, and the original program does not violate the policy during the guarded block, then the traces are equivalent on exit from the block. Moreover, the abstract states remain step-wise equivalent outside these blocks. This strategy reduces the vast majority of the search space that is unaffected by the IRM to a simple, linear scan that confirms that the IRM remains dormant outside these blocks (i.e., its state does not leak into the observable events exhibited by the rewritten code).

4.3 Transparency Verification Design

4.3.1 ActionScript Bytecode Core Subset

For expository simplicity, we express the verification algorithm in terms of a small (but Turing-complete), stack-based toy language that includes standard arithmetic operations, conditional and unconditional jumps, integer-valued local registers, and the special instructions listed in Fig. 4.1. The implementation described in Section 4.2 supports the full ActionScript bytecode language.

$\mathbf{api}_m n$	system API calls
$\mathbf{apptrace}_m n$	append to trace
$\mathbf{assert}_m n$	assert policy-adherence of event

Figure 4.1. Non-standard core language instructions

Instruction $\mathbf{api}_m n$ models a system API call, where m is a method identifier and n is the method's arity. Some API calls constitute observable events; these are modeled by an additional **apptrace** instruction that explicitly appends the API call event to the trace. Observable events can therefore be modeled as a macro $\mathbf{obsevent}_m n$ whose expansion is given in Fig. 4.2. In the rewritten code, the expansion simply appends the event to the trace and then performs the event. In the original code, the expansion additionally models the premise in Definition 4.2.2 that says that transparency is only an obligation when the original code does not violate the policy. The $\mathbf{assert}_m n$ instruction therefore asserts that the current flow is unreachable if exhibiting $\mathbf{apptrace}_m n$ at this point is a policy violation. This has the effect of pruning such flows from the search space.

The toy language models objects and their instance fields by reducing them to integer encodings, and exceptions are modeled as conditional branches in the typical way. A formal treatment of these is here omitted; their implementation in the transparency verifier is that of a standard abstract interpreter.

$$\begin{array}{rl|ll}
\multicolumn{2}{c|}{\textsc{Original}} & \multicolumn{2}{c}{\textsc{Rewritten}} \\
\mathbf{obsevent}_m n \equiv & \mathbf{assert}_m n & \mathbf{obsevent}_m n \equiv & \\
 & \mathbf{apptrace}_m n & & \mathbf{apptrace}_m n \\
 & \mathbf{api}_m n & & \mathbf{api}_m n
\end{array}$$

Figure 4.2. Semantics of the **obsevent** pseudo-instruction

4.3.2 Concrete and Abstract Machines

Concrete interpretation and abstract interpretation of ActionScript bytecode programs is expressed as the small-step operational semantics of a concrete and an abstract machine, respectively. Figure 4.3 defines a *concrete configuration* χ as a tuple consisting of a labeled bytecode instruction $L : i$, a concrete operand stack ρ, a concrete store σ, and a concrete trace of observable events τ. The store σ maps heap and local variables ℓ to their integer values. *Abstract configurations* $\hat{\chi}$ are defined similarly, except that abstract stacks, stores, and traces are defined over symbolic expressions instead of values. Expressions include integer-valued *meta-variables* $\hat{v}$ and return values $\mathbf{rval}_m(e_1{::}\cdots{::}e_n)$ of API calls. Meta-variables $\hat{s}$ and $\hat{t}$ denote entire abstract stacks and traces, respectively.

A *program* $P = (L, p, s)$ consists of a program entrypoint label L, a mapping p from code labels to program instructions, and a label successor function s that defines the destinations of non-branching instructions.

Since transparency verification involves bisimulating the original and instrumented programs, Fig. 4.4 extends the concrete and abstract configurations described above to *bisimulation machine* configurations. Each such configuration includes both an original and rewritten machine configuration. The abstract bisimulation machine additionally includes a constraint list ζ consisting of a conjunction of linear inequalities over expressions.

The concrete machine semantics are modeled after the ActionScript VM 2 (AVM2) semantics [Adobe Systems Incorporated, 2007]; the semantics of the special instructions of Fig. 4.1 are provided in Fig. 4.5. Relation $\chi \mapsto_P^n \chi'$ denotes n steps of concrete interpretation

L	(Code Labels)
i	(Instructions)
$P ::= (L, p, s)$	(Programs)
$p : L \to i$	(Instruction Labels)
$s : L \rightharpoonup L$	(Label Successors)
$m \in \mathbb{N}$	(Method Identifiers)
$n \in \mathbb{N}$	(Method Arities)
$\sigma : (r \uplus \ell) \rightharpoonup \mathbb{Z}$	(Concrete Stores)
$\rho ::= \cdot \mid x{::}\rho$	(Concrete Stacks)
$x \in \mathbb{Z}$	(Values)
$\tau ::= \epsilon \mid \tau\mathbf{api}_m(x_1{::}\cdots{::}x_n)$	(Concrete Traces)
$sys : \mathbb{N} \times \mathbb{Z}^* \to \mathbb{Z}$	(API Return Values)
$\mathfrak{a} : \tau \rightharpoonup \mathbb{N}$	(Security Automaton State)
$\chi ::= \langle L : i, \rho, \sigma, \tau \rangle$	(Concrete Config.)
$e ::= n \mid \hat{v} \mid e_1{+}e_2 \mid \ldots \mid$	(Symbolic Expressions)
$\mathbf{rval}_m(e_1{::}\cdots{::}e_n) \mid \hat{\mathfrak{a}}(\hat{\tau})$	
$\hat{v}, \hat{s}, \hat{t}$	(Value, Stack, & Trace Vars)
$\hat{\rho} ::= \cdot \mid \hat{s} \mid e{::}\hat{\rho}$	(Abstract Stacks)
$\hat{\sigma} : (r \uplus \ell) \to e$	(Abstract Stores)
$\hat{\tau} := \epsilon \mid \hat{t} \mid \hat{\tau}\mathbf{api}_m(e_1{::}\cdots{::}e_n)$	(Abstract Traces)
$\hat{\chi} ::= \langle L : i, \hat{\rho}, \hat{\sigma}, \hat{\tau} \rangle$	(Abstract Config.)
$\hat{\chi}_0 = \langle L_0 : p(L_0), \cdot, \hat{\sigma}_0, \epsilon \rangle$	(Initial Abstract Config.)

Figure 4.3. Concrete and abstract machine configurations

$$\zeta ::= \bigwedge_{i=1..n} t_i \quad (n \geq 1) \qquad \text{(Constraints)}$$

$$t ::= T \mid F \mid e_1 \leq e_2 \qquad \text{(Clauses)}$$

$$\Gamma = \langle \chi_{\mathcal{O}}, \chi_{\mathcal{R}} \rangle \qquad \text{(Concrete Interpreter States)}$$

$$\hat{\Gamma} = \langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta \rangle \qquad \text{(Abstract Interpreter States)}$$

$$\langle \mathcal{C}, \langle \chi_{\mathcal{O}_0}, \chi_{\mathcal{R}_0} \rangle, \mapsto_P^n \rangle \qquad \text{(Concrete Interpreter)}$$

$$\langle \mathcal{A}, \langle \hat{\chi}_{\mathcal{O}_0}, \hat{\chi}_{\mathcal{R}_0}, \zeta_0 \rangle, \rightsquigarrow_P^n \rangle \qquad \text{(Abstract Interpreter)}$$

Figure 4.4. Concrete and abstract bisimulation machines

$$\frac{x' = sys(m, x_1{::}x_2{::}\cdots{::}x_n)}{\langle L : \mathbf{api}_m n, x_1{::}x_2{::}\cdots{::}x_n{::}\rho, \sigma, \tau \rangle \mapsto \langle s(L) : p(s(L)), x'{::}\rho, \sigma, \tau \rangle}\text{(CAPI)}$$

$$\frac{\rho = x_1{::}x_2{::}\cdots{::}x_n{::}\rho'}{\langle L : \mathbf{apptrace}_m n, \rho, \sigma, \tau \rangle \mapsto \langle s(L) : p(s(L)), \rho, \sigma, \tau\mathbf{api}_m(x_1{::}x_2{::}\cdots{::}x_n) \rangle}\text{(CAppTrace)}$$

$$\frac{\rho = x_1{::}\cdots{::}x_n{::}\rho' \qquad \tau\mathbf{api}_m(x_1{::}\cdots{::}x_n) \in \mathcal{P}}{\langle L : \mathbf{assert}_m n, \rho, \sigma, \tau \rangle \mapsto \langle s(L) : p(s(L)), \rho, \sigma, \tau \rangle}\text{(CAssert)}$$

$$\frac{\chi_i \mapsto_1 \chi_i' \qquad \chi_j = \chi_j' \qquad i \neq j}{\langle \chi_{\mathcal{O}}, \chi_{\mathcal{R}} \rangle \mapsto \langle \chi_{\mathcal{O}}', \chi_{\mathcal{R}}' \rangle}\text{(CBisim)}$$

Figure 4.5. Concrete small-step operational semantics

of program P. Subscript P is omitted when the program is unambiguous, and when n is omitted it defaults to 1 step.

Rule CAPI models calls to the system API using an opaque function *sys* that maps method identifiers and arguments to return values. Any non-determinism in the system API is modeled by extending the prototypes of system API functions with additional arguments that read a non-deterministic portion of the initial state. Rule CBisim lifts the single-machine semantics to a bisimulation machine that non-deterministically chooses which machine to step next.

$$\frac{e' = \mathbf{rval}_m(e_1{::}e_2{::}\cdots{::}e_n)}{\begin{array}{l}\langle L : \mathbf{api}_m n, e_1{::}e_2{::}\cdots{::}e_n{::}\hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle \rightsquigarrow \\ \qquad \langle s(L) : p(s(L)), e'{::}\hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle, T\end{array}} \text{(AAPI)}$$

$$\frac{\hat{\rho} = e_1{::}\cdots{::}e_n{::}\hat{\rho}'}{\begin{array}{l}\langle L : \mathbf{apptrace}_m n, \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle \rightsquigarrow \\ \qquad \langle s(L) : p(s(L)), \hat{\rho}, \hat{\sigma}, \hat{\tau}\mathbf{api}_m(e_1{::}\cdots{::}e_n)\rangle, T\end{array}} \text{(AAppTrace)}$$

$$\frac{\zeta = \big(0 \le \hat{\mathfrak{a}}(\hat{\tau}\mathbf{api}_m(e_1{::}\cdots{::}e_n))\big)}{\langle L : \mathbf{assert}_m n, \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle, \zeta} \text{(AAssert)}$$

$$\frac{\hat{\chi}_{\mathcal{O}} \subseteq \hat{\chi}'_{\mathcal{O}} \qquad \hat{\chi}_{\mathcal{R}} \subseteq \hat{\chi}'_{\mathcal{R}} \qquad \zeta \Rightarrow \zeta'}{\langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta\rangle \rightsquigarrow \langle \hat{\chi}'_{\mathcal{O}}, \hat{\chi}'_{\mathcal{R}}, \zeta'\rangle} \text{(Abstraction)}$$

$$\frac{\hat{\chi}_i \rightsquigarrow \hat{\chi}'_i, \zeta' \qquad \hat{\chi}_j = \hat{\chi}'_j \qquad i \neq j}{\langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta\rangle \rightsquigarrow \langle \hat{\chi}'_{\mathcal{O}}, \hat{\chi}'_{\mathcal{R}}, \zeta \wedge \zeta'\rangle} \text{(ABisim)}$$

Figure 4.6. Abstract small-step operational semantics

Figure 4.6 gives the corresponding semantics for abstract interpretation. Each step $\hat{\chi} \rightsquigarrow \hat{\chi}', \zeta$ of abstract interpretation yields both a new configuration $\hat{\chi}'$ and a list ζ of new constraints. These are conjoined into the master list of constraints by rule ABisim.

Rule AAPI uses expression $\mathbf{rval}_m(\cdots)$ to abstractly denote the return value of API call m. Rule AAssert introduces a new constraint that asserts that appending API call m to the current trace yields a policy-adherent trace. The constraint uses expression $\hat{\mathfrak{a}}(\hat{\tau}')$, which abstractly denotes the *security automaton state* resulting from trace $\hat{\tau}'$. Rule Abstraction allows the abstract interpreter to discard information at any point by abstracting the current state. This facilitates pruning the search space in response to hints from the invariant-generator. Discarding too much information can result in conservative rejection, but it never results in incorrect acceptance of non-transparent code.

4.3.3 Verification Algorithm

Algorithms 7 and 8 present the main transparency verification algorithm in terms of the bisimulating abstract interpreter described in Section 4.3.2. Algorithm 8 verifies an individual

Algorithm 7: Verification

```
Input: $Cache = \{\}$, $Horizon = \{\hat{\Gamma}_0\}$
Output: $Accept$ or $Reject$
1: while $Horizon \neq \emptyset$ do
2:     $\hat{\Gamma} \leftarrow choose(Horizon)$
3:     $S_{\hat{\Gamma}} \leftarrow VerificationSingleCodePoint(\hat{\Gamma})$
4:     if $S_{\hat{\Gamma}} = Reject$ then return $Reject$
5:     $Cache \leftarrow Cache \cup \{\hat{\Gamma}\}$
6:     $Horizon \leftarrow (Horizon \cup S_{\hat{\Gamma}}) \setminus Cache$
7: end while
8: return $Accept$
```

abstract state of the bisimulation, and Algorithm 7 calls it as a subroutine to explore and verify transparency for all abstract states in all reachable control flows. We begin with a description of Algorithm 7.

Algorithm 7 takes as input a cache of previously explored abstract states and a horizon of unexplored abstract states. Upon successful verification of all control flows, it returns *Accept*; otherwise it returns *Reject*. It begins by drawing an arbitrary unexplored bisimulation state $\hat{\Gamma}$ from the *Horizon* (line 2) and passing it to Algorithm 8. Algorithm 8 returns a set $S_{\hat{\Gamma}}$ of abstract states where bisimulation must continue in order to verify all control-flows proceeding from $\hat{\Gamma}$ (line 3). Every state of $S_{\hat{\Gamma}}$ that is not already in the *Cache* is added to the *Horizon* (line 6). Verification concludes when all the states in *Horizon* have been explored.

Algorithm 8 takes an abstract state $\hat{\Gamma}$ as input. It begins by asking the invariant-generator for a hint (line 1), consisting of: (1) a new (possibly more abstract) state $\hat{\Gamma}_H$ for $\hat{\Gamma}$, (2) a finite, *generalized post-dominating set* $D_{\hat{\Gamma}}$ for $\hat{\Gamma}$ whose members are all trace-equivalent code points, and (3) a stepping-bound n. A set S of abstract states is said to be generalized post-dominating for $\hat{\Gamma}$ if every complete control-flow that includes $\hat{\Gamma}$ later includes at least one member of S [Gupta, 1992]. In our case, the complete flows are the infinite ones (since termination is modeled as an infinite stutter state). A code point is said to be trace-equivalent if its invariant implies that the original and rewritten traces are equal. The stepping bound n is an upper bound on the number of steps required to reach any state in $D_{\hat{\Gamma}}$ from $\hat{\Gamma}_H$.

Algorithm 8: VerificationSingleCodePoint

```
Input: $\hat{\Gamma} = \langle\langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1\rangle, \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2\rangle, \zeta\rangle$
Output: $S_{\hat{\Gamma}}$ or $\mathit{Reject}$
 1: $(\hat{\Gamma}_H, D_{\hat{\Gamma}}, n) \leftarrow \mathit{InvariantGen}(\hat{\Gamma})$
 2: $\mathit{SatValue} \leftarrow \mathit{ModelCheck}(\hat{\Gamma}, \hat{\Gamma}_H)$
 3: if $\mathit{SatValue} = \mathit{Reject}$ then return $\mathit{Reject}$
 4: $S_{\hat{\Gamma}} \leftarrow \mathit{AbsIn}^n(\{\hat{\Gamma}_H\}, D_{\hat{\Gamma}})$
 5: if $\mathit{labels}(S_{\hat{\Gamma}}) \not\subseteq D_{\hat{\Gamma}}$ then return $\mathit{Reject}$
 6: for all $\hat{\Gamma}' = \langle\hat{\chi}_1, \hat{\chi}_2, \zeta'\rangle \in S_{\hat{\Gamma}}$ do
 7:     $\langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1\rangle \leftarrow \hat{\chi}_1$
 8:     $\langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2\rangle \leftarrow \hat{\chi}_2$
 9:     $\hat{\chi}'_1 \leftarrow \langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{t}_1\rangle$
10:     $\hat{\chi}'_2 \leftarrow \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{t}_2\rangle$
11:     $\mathit{SatValue} \leftarrow \mathit{ModelCheck}(\hat{\Gamma}', \langle\hat{\chi}'_1, \hat{\chi}'_2, \zeta' \wedge (\hat{t}_1 = \hat{t}_2)\rangle)$
12:     if $\mathit{SatValue} = \mathit{Reject}$ then return $\mathit{Reject}$
13: end for
14: return $S_{\hat{\Gamma}}$
```

Set $D_{\hat{\Gamma}}$ and bound n therefore constitute a witness that proves that even if state $\hat{\Gamma}$ is not trace-equivalent, the traces become equivalent within at most n steps of computation.

The hint obtained in line 1 is not trusted; it is typically a hint provided by the rewriter itself or some untrusted third party. It must therefore be verified. To do so, model-checking first confirms that $\hat{\Gamma}_H$ is a sound abstraction of $\hat{\Gamma}$ according to the Abstraction rule of the operational semantics (see Fig. 4.6). Next, abstract interpretation for n steps from $\hat{\Gamma}$ is applied to confirm post-dominance of $D_{\hat{\Gamma}}$. Function *AbsIn* in line 4 is defined by

$$\begin{aligned}\mathit{AbsIn}(S, E) = &\{\hat{\Gamma}' \mid \hat{\Gamma} \in S, \mathit{labels}(\hat{\Gamma}) \notin E, \hat{\Gamma} \rightsquigarrow \hat{\Gamma}'\} \cup \\ &\{\hat{\Gamma} \in S \mid \mathit{labels}(\hat{\Gamma}) \in E\}\end{aligned}$$

where $\mathit{labels}(\langle L_1{:}\ldots\rangle, \langle L_2{:}\ldots\rangle) = (L_1, L_2)$ extracts the code labels of an abstract state. Function $\mathit{AbsIn}(S, E)$ therefore abstract interprets all states in S for one step, except that states already at end code-points in E are not interpreted further. Finally, the model-checker is applied again to confirm trace-equivalence of all members of $S_{\hat{\Gamma}}$ (line 12). If successful, set $S_{\hat{\Gamma}}$ is returned.

Algorithm 9: ModelCheck

```
Input: $\hat{\Gamma} = \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$, $\hat{\Gamma}' = \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \rangle$
Output: Accept or Reject
1: $\zeta_U \leftarrow \mathit{Unify}(\hat{\Gamma}, \hat{\Gamma}')$
2: if $\zeta_U = \mathit{Fail}$ then return Reject
3: $\mathit{SatValue} \leftarrow \mathit{CLP}(\zeta \wedge \neg\zeta' \wedge \zeta_U)$
4: if $\mathit{SatValue} = \mathit{False}$ then
5:     return Accept
6: else
7:     return Reject
8: end if
```

4.3.4 Model-Checking

Verification of abstract states suggested by the invariant-generator, pruning of policy-violating flows, and verification of trace-equality are all reduced by Algorithm 8 to proving implications of the form $A \Rightarrow B$. These are proved by the two-stage model-checking procedure in Algorithm 9, consisting of unification followed by linear constraint solving.

Unification. Each abstract configuration $\hat{\chi}$ can be viewed as a set of equalities that relate the various configuration components to their values. Many of these equalities relate entire structures; for example, each operand stack is an ordered list of expressions. Given two abstract states $\hat{\Gamma} = \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$ and $\hat{\Gamma}' = \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \rangle$, the model-checker first uses Prolog unification to mine all structural equalities in all of the state components for equalities over their contents. For example, if $\hat{\rho}_1 = e_1{::}\hat{s}$ and $\hat{\rho}'_1 = e'_1{::}e'_2{::}\hat{s}'$, then unification infers $e_1 = e'_1$ and $\hat{s} = e'_2{::}\hat{s}'$. If unification fails, the model-checker rejects.

Since constraint lists ζ and ζ' contain only value inequalities, any structural equalities inferred by the unification are irrelevant for falsifying the constraint lists. They can therefore be safely dropped once unification succeeds. In the example above, constraint $\hat{s} = e'_2{::}\hat{s}'$ is discarded. The result is a set of purely non-structural equalities.

Linear Constraint Solving. Once both states have been reduced to a conjunction ζ_U of non-structural equalities, the model-checker applies constraint logic programming (CLP) to verify that $\neg(\zeta \wedge \zeta_U \Rightarrow \zeta')$ is not satisfiable. That is, it confirms that under the hypothesis ζ_U that $\hat{\Gamma}$ and $\hat{\Gamma}'$ abstract the same code point, there is no instantiation of the free variables that falsifies $\zeta \Rightarrow \zeta'$.

The constraints that populate lists ζ arise from four major sources: conditional branches, observable events in the original code, trace-equality checks, and hints provided by the invariant-generator. Each plays an important role in practical transparency verification, so we discuss each.

Conditional branches introduce constraints of the form $e_1 \leq e_2$ to the abstract state. These are important for verifying that some outgoing branches from IRM conditional guards are only traversed when the original code violates the policy; the transparency obligation is therefore waived for such flows. This permits IRMs to implement transparency-violating intervention code as long as it does not observably affect non-violating runs of the program.

Observable events in the original code introduce constraints of the form $0 \leq \hat{\mathfrak{a}}(\hat{\tau})$ (see rule AAssert of Fig. 4.6) which assert that the security automaton that encodes the policy is in a well-defined state (i.e., it has not rejected the original program). This prunes flows in which the original program violates security, avoiding conservative rejection of rewritten code that intervenes appropriately.

Trace-equality checks are performed by line 11 of Algorithm 8, which queries the model-checker with constraints of the form $\hat{t}_1 = \hat{t}_2$. These inquire whether equivalence of two traces is provable from information that is known about the current bisimulation state. If the constraint is falsifiable, the verifier conservatively rejects.

The untrusted invariant-generator may also introduce constraints that encode loop invariants relevant to proving transparency. This is critical for tractably exploring the full state space and keeping the TCB small. Invariant-generation is discussed in greater detail in the next Section 4.3.5.

4.3.5 Invariant Generation

Recall from Section 4.3.3 that for every reachable code point (L_1, L_2) in the bisimulation state space, the verifier requires an (untrusted) hint consisting of: (1) an invariant for (L_1, L_2) in the form of an abstract bisimulation machine state, (2) a finite, generalized post-dominating set for (L_1, L_2) whose members are all trace-equivalent code points, and (3) a stepping-bound n.

These hints may come from any untrusted source, since they are independently verified by the trusted model-checker. This is important for both verifier generality and TCB minimization. General-purpose invariant-generation is well known to be extraordinarily difficult, so it is unlikely that any one invariant-generation strategy suffices for all IRMs. Shifting invariant-generation outside the TCB addresses this problem by allowing it to be tailored to the specific rewriting algorithm without re-proving correctness of the verification algorithm for each possible invariant-generator.

In this section we outline a strategy for invariant-generation that suffices for our rewriting system, and that can be used as a basis for transparency verification of many other IRM systems that adopt similar instrumentation approaches. Our approach capitalizes on the fact that most rewriters leave large portions of the original code unchanged. They implement IRMs as collections of small code blocks that guard security-relevant operations. Other additions typically include new classes with helper methods that maintain and track security automaton state as private fields.

The set of IRM-instrumented code labels is exposed to our invariant-generator as a set *Marked* of *code marks* that distinguish code regions that implement the IRM from code that has been left relatively unchanged during rewriting. Marked regions include IRM guard code and the security-relevant instructions it guards, but not IRM intervention code that responds to impending violations. (Interventions remain unmarked with the expectation that the verifier considers them unreachable under the assumption that the original code does not

Algorithm 10: GenericInvariant

```
Input: $\hat{\chi}_1 = \langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1 \rangle, \hat{\chi}_2 = \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2 \rangle, \zeta$
 1: choose fresh meta-variables $\hat{v}_\ell$, $\hat{v}'_\ell$, $\hat{s}$, $\hat{s}'$, $\hat{t}$, and $\hat{t}'$
 2: $\hat{\sigma} \leftarrow \{(\ell, \hat{v}_\ell) \mid \ell \in \hat{\sigma}_1^{\leftarrow}\}$
 3: $\hat{\sigma}' \leftarrow \{(\ell, \hat{v}'_\ell) \mid \ell \in \hat{\sigma}_2^{\leftarrow}\} \cup \{(r, \hat{\sigma}_2(r))\}$
 4: $\hat{\chi} \leftarrow \langle L_1 : i_1, \hat{s}, \hat{\sigma}, \hat{t} \rangle$
 5: $\hat{\chi}' \leftarrow \langle L_2 : i_2, \hat{s}', \hat{\sigma}', \hat{t}' \rangle$
 6: $\zeta = (\hat{s}{=}\hat{s}') \wedge \left(\bigwedge_{\ell \in \hat{\sigma}^{\leftarrow} \cap \hat{\sigma}'^{\leftarrow}} \hat{v}_\ell{=}\hat{v}'_\ell\right) \wedge (r{=}\hat{\mathfrak{a}}(\hat{t}')) \wedge (\hat{t}{=}\hat{t}')$
 7: return $\mathbf{I} = \langle \hat{\chi}, \hat{\chi}', \zeta \rangle$
```

violate the policy.) The invariant-generator then returns a different kind of hint depending on whether the bisimulation state is within a marked region.

Outside marked regions it uses Algorithm 10 to generate a hint that asserts that the original and rewritten machines are step-wise equivalent. That is, all original and rewritten state components are equal except for state introduced by the IRM and that therefore does not exist in the original code. It additionally asserts that reified state variables introduced by the IRM accurately encode the current security state; this is captured by clause $(r = \hat{\mathfrak{a}}(\hat{t}_{\mathcal{R}}))$ in line 6. This property is necessary for proving that interventions are unreachable, since the verifier must be able to conclude that guard code that tests reified state makes accurate inferences about the abstract security state.

Within marked regions, the invariant-generator uses the last half of Algorithm 11, which asserts that trace-equivalence is restored once bisimulation exits the marked region. To prove that execution does eventually exit the marked region, line 5 uses abstract interpretation to find the point at which each control-flow exits *Marked*. IRMs implementing non-trivial loops outside of interventions may cause this step to conservatively fail.

While the invariant-generation algorithm presented here is specific to our instrumentation algorithm, it can be adapted to suit other similar instrumentation algorithms by replacing constraint $(r = \hat{\mathfrak{a}}(\hat{t}_{\mathcal{R}}))$ in Algorithm 10 with a different constraint that models the way in which the IRM reifies the security state. Similarly, appeals to set *Marked* can be replaced

Algorithm 11: InvariantGen

```
Input: $\hat{\chi}_1 = \langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1 \rangle, \hat{\chi}_2 = \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2 \rangle, \zeta$
Output: $\hat{\Gamma}_H$,$D_{\hat{\Gamma}}$,$n$
1: if $L_2 \in \mathit{Marked}$ then
2:     return $(\mathit{GenericInvariant}(\hat{\chi}_1, \hat{\chi}_2), \{(L_1, L_2)\}, 1)$
3: else
4:     $\hat{\Gamma} \leftarrow \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$
5:     $n \leftarrow \min\{n \mid \mathit{AbsIn}^n(\{\hat{\Gamma}\}, \mathit{Marked}) =$
                       $\mathit{AbsIn}^{n+1}(\{\hat{\Gamma}\}, \mathit{Marked})\}$
6:     return $(\hat{\Gamma}, \mathit{labels}(\mathit{AbsIn}^n(\{\hat{\Gamma}\}, \mathit{Marked})), n)$
7: end if
```

```
     L1: push "http:// ..."
×    L2:    get c
×    L3:    iflt 100, L5    // if c ≤ 100 goto L5
     L4:    call exit
×    L5: call NavigateToURL
×    L6:    get c
×    L7:    push 1
×    L8:    add
×    L9:    set c
    L10: jmp L1
```

Figure 4.7. An IRM that prohibits more than 100 URL navigations

with alternative logic that identifies code points where the transparency invariant is restored after the IRM has completed any maintenance associated with security-relevant events.

4.3.6 A Verification Example

To illustrate, we briefly describe transparency verification of the simple pseudo-bytecode listing in Figure 4.7, which implements an IRM that prohibits more than 100 calls to security-relevant method `NavigateToURL`. Indented lines are instructions in-lined by the IRM; non-indented lines are from the original code. Lines with an $\times$ are those in the *Marked* set described in Section 4.3.5. This IRM tracks the number of calls to `NavigateToURL` in global field c. (Real IRMs are typically much more complex, but we use this simplified example for clarity.)

The abstract interpreter begins exploring the cross-product space from point $(L1, L1)$, where both original and rewritten programs are at line 1. The initial state asserts equivalence of all state components except c, which does not exist in the original code and is initialized to 0 in the rewritten code. The (untrusted) invariant-generator suggests a hint that abstracts this to a clause of the form $c = \hat{\mathfrak{a}}(\hat{t})$, where meta-variable $\hat{t}$ denotes the current trace. This invariant recommends that the only information necessary at line 1 to infer transparency is that c correctly reflects the abstract security state. Since initially $\hat{t} = \hat{\tau} = \epsilon$ in both machines, the verifier confirms that this abstraction hint is sound and uses it henceforth as an invariant for point $(L1, L1)$.

The invariant-generator next supplies a post-dominating set $\{(L10, L10)\}$ and stepping bound 8, which asserts that all realizable flows from $(L1, L1)$ take both machines to line 10 within 8 steps. The verifier confirms this by non-deterministically interpreting all paths from $(L1, L1)$ for 8 steps. When interpretation reaches the conditional at line 3, clause $c = \hat{\mathfrak{a}}(\hat{t})$ is critical for inferring that line 4 is unreachable when the original code satisfies the policy. Specifically, the policy-adherence assumption yields constraint $\hat{\mathfrak{a}}(\hat{\tau}) < 100$ after line 5, which contradicts negative branch condition $c \geq 100$ introduced by line 4 of the rewritten code when $c = \hat{\mathfrak{a}}(\hat{\tau})$.

Once the interpreter reaches point $(L10, L10)$, the invariant-generator supplies the same abstract state as it did for line 1. That is, the state opaquely asserts that all common state components (including traces) are equal, and reified state c equals abstract state $\hat{\mathfrak{a}}(\hat{t})$. The linear constraint solver confirms that the incremented c (line 8) matches the incremented state $\hat{\mathfrak{a}}(\hat{t}\,\mathbf{api}_{\text{NavigateToURL}})$, and therefore accepts the new invariant. Abstract interpreting for an additional step, it confirms that this matches the loop invariant supplied for line 1, and accepts the program-pair as transparent.

Theorem 4.3.1 (Transparency). *If Algorithm 7 terminates and returns Accept, the program-pair is transparent.*

Proof. This theorem is proved in our technical report [Sridhar et al., 2012] □

4.4 Evaluation

Our implementation of the transparency verification algorithm detailed in Section 4.2 targets the full ActionScript bytecode language. It consists of 2500 lines of Prolog for 32-bit Yap 6.2 that parses and verifies pairs of Shockwave Flash File (SWF) binary archives. We use YAP CLP(R) [Jaffar et al., 1992] for constraint solving and Yap's tabling for memoization.

IRM instrumentation is accomplished via a collection of small binary-to-binary rewriters that augment untrusted ActionScript code with security guards in accordance with a security policy. For ease of implementation, each rewriter is specialized to a particular policy class. For example, one rewriter enforces *resource bound* policies that limit the number of accesses to policy-specified system API functions per run. It augments untrusted code with counters that track accesses, and halts the applet when an impending operation would exceed the bound. Each rewriter is accompanied by an invariant-generator as described in Section 4.2. Rewriters were each about 200 lines of Prolog (not including parsing) and invariant-generaters were about 100 lines each.

To demonstrate the versatility of our transparency verifier, rewriters in our framework performed localized binary optimizations during rewriting when convenient. For example, when original code followed by IRM code formed a sequence of consecutive conditional branches, the entire sequence (including the original code) was replaced with an ActionScript multi-way jump instruction (`lookupswitch`). Certifying transparency of the instrumented code therefore required the verifier to infer semantic equivalence of these transformations.

When implementing our IRMs we found the transparency verifier to be a significant aid to debugging. Bugs that we encountered included IRMs that fail transparency when in-lined into unusual code that overrides IRM-called methods (e.g., toString), IRMs that could throw uncaught exceptions (e.g., null pointer) in rare cases, IRMs that inadvertently trigger class initializer code that contains an observable operation, and broken IRM instructions that corrupt a register or stack slot that flows to an observable operation. All of these were immediately detected by the verifier, greatly facilitating debugging.

We applied our prototype framework to rewrite and verify the resulting transparency of numerous real-world Flash ads drawn from public web sites. The results are summarized in Table 4.4. For each ad, the table columns report the policy type, bytecode size before and after rewriting, the number of methods in the original code, and rewrite and verification times. All tests were performed on a Lenovo Z560 notebook computer running Windows 7 64-bit with an Intel Core I5 M480 dual core processor, 2.67 GHz processor speed, and 4GB of memory.

Except for `HeapSprayAttack` (a synthetic attack discussed below) all tested ads were being served by public web sites when we collected them. Some came from popular business and ecommerce websites, but the more obtrusive ones with potentially undesirable actions tended to be hosted by less reputable sites, such as adult entertainment and torrent download pages. Potentially undesirable actions included unsolicited URL redirections, large pop-up expansions, tracking cookies, and excessive memory usage. However, ad complexity was not necessarily indicative of maliciousness; some of the most complex ads were benign. For example, `wine` implements a series of interactive menus showcasing wines and ultimately offering navigation to the seller's site. Others function as support code for flash video players and cookie maintenance.

All programs are classified into one of four case study classes:

Capping URL Navigations. One resource bound policy we enforced restricts the number of times a Flash applet may navigate away from the hosting site. This helps to prevent unwanted chains of pop-up windows. The IRM enforces the policy by counting calls to the `NavigateToURL()` system API function. When an impending call would exceed the bound,

Table 4.1. Experimental Results

Program	**Policy**	**File Size** (KB) old	new	**No. Meth.**	**Rewrite Time** (ms)	**Verif. Time** (ms)
`adult1`	`ResBnds`	1	2	4	<1	<1
`adult2`		18	18	102	127	1201
`atmos`		1	1	6	<1	<1
`att`		22	22	147	156	1434
`ecls`		2	3	6	16	<1
`eco`		2	3	6	<1	16
`flash`		3	4	12	<1	62
`fxcm`		2	2	12	16	16
`gm`		21	22	142	157	1245
`gucci`		2	2	6	15	16
`iphone`		2	2	6	<1	<1
`IPLad`		2	2	15	31	15
`jlopez`		17	17	151	95	560
`lowes`		34	34	181	218	16549
`men1`		33	34	237	203	3757
`men2`		40	40	270	297	4964
`prius`		71	71	554	516	10359
`priusm`		70	71	542	468	9951
`sprite`		34	34	324	234	3075
`utv`		21	21	155	151	1171
`verizon1`		3	4	25	<1	37
`verizon2`		3	3	12	31	15
`weightwatch`		4	4	34	47	47
`wines`		185	185	926	904	35926
`expandall`	`NoExpands`	3	4	17	47	79
`cookie`	`NoCookieSet`	3	3	8	31	16
`CookieSet`		1	1	4	<1	<1
`HeapSprAttk`	`NoHeapSpray`	1	1	4	15	15

the call is suppressed at runtime by a conditional branch. To verify transparency of the resulting IRM, the verifier proves that such branches are only reachable in the event of a policy violation by the original code.

Bounding Cookie Storage. For another resource bounds policy, we limited the number of cookie creations per ad. This was achieved by guarding calls to the `SetCookie()` API function. Impending violations cause the IRM to prematurely halt the applet.

Preventing Pop-up Expansions. Some Flash ads expand to fill a large part of the web page whenever the user clicks or mouses over the ad space. This is frequently abused for click-jacking. Even when ad clicks solicit purely benign behavior, many web publishers and users regard excessive expansion as essentially a denial-of-service attack upon the embedding page. There is therefore high demand for a means of disabling it. Our expansion-disabling policy does so by denying access to the `GoToAndPlay()` system API function.

Heap Spray Attacks. *Heap spraying* is a technique for planting malicious payloads by allocating large blocks of memory containing sleds to dangerous code. Cooperating malware (often written in an alternative, less safe language) can then access the payload to do damage, for example by exploiting a buffer overrun to jump to the sled. By separating the payload injector and exploit code in different applications, the attack becomes harder to detect.

ActionScript has been used as a heap spraying vehicle in several past attacks [FireEye, 2009]. The attack code typically allocates a large byte array and inserts the payload into it one byte at a time. This implementation makes it difficult to reliably detect the payload's signature via purely static inspection of the ActionScript binary.

To inhibit heap sprays, we enforced a policy that prevents a large (policy-specified) number of byte-write operations from being performed by a Flash ad. We then implemented a heap spray (`HeapSprAttk`) and verified that the IRM successfully prevented the attack. Applying the policy to all other ads in Table 4.4 resulted in no behavioral changes, as confirmed by the transparency verifier.

4.5 Transparency for x86 Rewriters

This is only preliminary work towards the more difficult task of statically proving behavioral equivalence between an original and instrumented version of an x86 binary. IRMs for type-safe bytecode languages can be efficiently implemented without permanently modifying pre-existing program variables. In contrast, efficient implementation of IRMs for native code binaries requires permanently modifying computed jump arguments in order to tame their much less restricted control-flows. Proving transparency for native code IRMs thus raises the significant additional challenge of proving that these permanent state changes do not result in observable behavioral changes by the rewritten code (assuming the original program is policy-adherent). Future work should consider extending the work presented in this chapter to verify transparency of native code IRMs.

CHAPTER 5

RELATED WORK

Numerous topics are closely related to this work, as well as work that has attempted to solve similar problems with different approaches. This chapter will first discuss the different approaches to disassembly in Section 5.1, followed by a discussion of binary rewriting approaches in Section 5.2 and behavioral equivalence of rewriters in Section 5.3.

5.1 Disassembly

Disassemblers of x86 binaries can be broken into two classes: *static* and *dynamic*. Static disassembly, as discussed in Section 2.2, involves classifying a byte sequence as code or data in order to discover the underlying semantics of the binary. Since many binary analysis and instrumentation tools rely on static disassembly, including both Reins and Stir, more accurate disassemblies translate to shortened analysis times and more accurately rewritten binaries.

Existing static disassemblers mainly fall into three categories: linear sweep disassemblers, recursive traversal disassemblers, and the hybrid approach. The GNU utility *objdump* [GNU Project, 2012] is a popular example of the linear sweep approach. It starts at the beginning of the text segment of the binary to be disassembled, decoding one instruction at a time until everything in executable sections is decoded. This type of disassembler is prone to errors when code and data bytes are interleaved within some segments. Such interleaving is typical of almost all production-level Windows binaries.

IDA Pro [Hex-Rays, 2012, Eagle, 2008] follows the recursive traversal approach. Unlike linear sweep disassemblers, it decodes instructions by traversing the static control flow of the program, thereby skipping data bytes that may punctuate the code bytes. However, not all control flows can be predicted statically. When the control flow is constructed incorrectly, some reachable code bytes are missed, resulting in disassemblies that omit significant blocks of code.

The hybrid approach [Schwarz et al., 2002] combines linear sweep and recursive traversal to detect and locate disassembly errors. The basic idea is to disassemble using the linear sweep algorithm and verify the output using the recursive traversal algorithm. While this helps to eliminate some disassembly errors, in general it remains prone to the shortcomings of both techniques. That is, when the sweep and traversal phases disagree, there is no clear indication of which is correct; the ambiguous bytes therefore receive an error-prone classification.

Static disassembly for obfuscated binaries [Krügel et al., 2004] attempts to discover all of the used execution paths in a binary by simulating the call stack statically. Since most mainstream disassemblers do not attempt to find more than a single execution path, this approach performs very well on obfuscated binaries compared to disassemblers like objdump and IDA Pro.

The Jakstab fully configurable binary analysis platform [Kinder and Veith, 2008, Kinder et al., 2009, Kinder and Kravchenko, 2012, Kinder, 2012] is a recent effort to overcome these historic shortcomings by statically resolving computed jump destinations to construct accurate control-flow graphs. The platform implements multiple rounds of disassembly interleaved with dataflow analysis. In each round, the output assembly instructions are translated to an intermediate representation, from which the platform builds a more accurate control-flow graph for dataflow analysis. The results from the dataflow analysis are then used to resolve computed jump targets. This iterative approach exhibits superior accuracy over commercial tools like IDA Pro [Hex-Rays, 2012]. However, Jakstab depends on a fixed-point iteration algorithm that is worst-case exponential in the size of the binary being analyzed. It therefore has only been successfully applied to relatively small binaries (e.g., drivers) and does not scale well to full-sized COTS binaries, which are often ten or one hundred times larger. For example, Jakstab requires almost 40 minutes[1] to disassemble a 100K Windows floppy driver [Kinder and Veith, 2010].

Our recent machine learning- and data mining-based approach to the disassembly problem reported in Section 2.4 [Wartell et al., 2011] avoids error-prone control-flow analysis heuristics in favor of a three-phase approach: First, executables are segmented into subsequences of bytes that constitute valid instruction encodings as defined by the architecture [Intel Corporation, 2012]. Next, a language model is built from the training corpus with a statistical data model used in modern data compression. The language model is used to classify the segmented subsequence as code or data. Finally, a set of pre-defined heuristics refines the classification results. The experimental results demonstrate substantial improvements over IDA Pro's traversal-based approach. However, it has the disadvantage of high memory usage due to the large statistical compression model. This significantly slows the disassembly process relative to simple sweep and traversal disassemblers.

Machine-learning has also been applied to statically identify errors in disassembly listings [Krishnamoorthy et al., 2009]. Incorrect disassemblies are typically statistically different from correct disassemblies. Based on this observation, a decision tree classifier can be trained using a set of correct and incorrect disassemblies. The classifier is then used to detect errors such as extraneous opcodes and operands, as well as nonexistent branch target addresses. The experimental results demonstrate that the decision tree classifiers can correctly identify the majority of the disassembly errors in test files while returning relatively few false positives.

The alternative, dynamic disassembly, is guaranteed to be sound (only includes reachable flows) but not complete (including all reachable flows) and involves following as many

[1]This high runtime is due in part to Jakstab's strategy of estimating the possible targets of each individual computed jump, rather than merely the set of all reachable code bytes. The work presented in Chapter 3 does not need this extra information to successfully transform untrusted binaries, so our disassembler avoids computing it.

executions of a program as can be created. However the inherent shortcomings of this approach are code coverage and the necessity of executing the binary. Given a protection system charged with analyzing incoming binaries in real-time, running each incoming program to completion in a closed environment (e.g., a VM sandbox) is not feasible in the presence of real-time constraints. In addition, most binaries do not execute all of the code in their code sections on any one run, and thus large chunks of the binary are treated as data even though valid execution paths through them exist. Since the focus of this dissertation is a static approach to binary rewriting, we do not include dynamic disassembly techniques as related work.

5.2 Binary Rewriting

Most binary rewriting technologies fall into 4 categories:

1. rewriters requiring recompilation of the binary,
2. rewriters that instrument at runtime using a virtual machine,
3. rewriters that use debugging info or link-time information, and
4. rewriters that can be used on arbitrary binaries, but with significant constraints.

5.2.1 Recompilation

Using compile-time information to redesign the binary instruction sequence is the most common solution. This can be done as part of the compiler, changing the translation from source code to x86 binary, or it can be done using link-time information, using function and symbol information to perform the rewriting process. Instrumenting x86 binaries this way involves modifying a compiler or intercepting information at link time. However, working at the compiler level assumes code-producer cooperation, forfeiting arbitrary binary support.

Another alternative to securing binaries is certified compilation: a compiler that provides a machine checkable proof of security (type safety, memory safety or control flow safety) in conjunction with the binary. However, certified compilation is difficult depending on the programming language.

Compiler Modification

The bulk of works that use recompilation as a means of binary rewriting are focused on Software Fault Isolation (SFI). Efficient SFI [Wahbe et al., 1993] is the earliest work in SFI, focusing on enforcing SFI for DEC-MIPS and DEC-ALPHA. SFI is enforced via sandboxing, which isolates an executing process such that it is restricted to a small allowed address space, restricting control flows, reads, and writes to its confines.

MiSFIt [Small and Seltzer, 1996] was developed soon after and closely resembles the work done recently in PittSFIeld [McCamant and Morrisett, 2006]. Both work with a modified version of GNU's GCC compiler, but the guards used to implement SFI in each system differ greatly. Where MiSFIt inserts guards in front of all potentially SFI violating instructions causing very high overhead, PittSFIeld breaks a binary into 16 byte chunks which allows for much shorter guards and much smaller overhead using clever masks.

Google researchers followed up on this work in NaCl [Yee et al., 2009]. NaCl implements SFI almost identically to PittSFIeld, with its main contribution being its method of propagation. NaCl works as a plugin for Google's Chrome browser, allowing code-producers the opportunity to run their code in a sandboxed area of the browser's virtual address space. An API is also available that allows code-producers to directly display their program within the browser window, effectively turning native code into web browser plugins or services.

Link-time Interception

Alternatively, ATOM [Eustace and Srivastava, 1995], PLTO [Schwarz et al., 2001], and Diablo [Sutter et al., 2005] were developed as a binary transformation tools—ATOM for Alpha binaries, PLTO for x86 binaries on Red Hat Linux, and Diablo for multiple architectures. Rather than modifying the compiler, they use relocation information and the map provided to a linker as information necessary to aid in instrumentation. This approach still requires recompilation since arbitrary binaries do not provide link-time information, since they have essentially already been “linked”.

Certified Compilation

CompCERT [Leroy, 2009] is a project focusing on certified compilation for the C++ programming language. Though quite successful, it still does not support the full range of C++ functionality, mainly because of the difficulty of proving safety properties on the x86 instruction set.

5.2.2 Use of Debug Information

Microsoft Research created the binary rewriting system known as Vulcan [Srivastava et al., 2001] which allows instrumentation of binaries on different architectures. Vulcan is not involved in compilation, instead disassembling a compiled executable to an intermediary representation that can then be re-assembled in a new binary. This allows for cross-architecture rewriting. However, in order to perform its conversion to intermediary language, Vulcan requires a program database file (PDB), which provides assembly procedure names, symbol table information, variable type information, and more, to aid in disassembly. Most companies do not provide a PDB file with their software since it could be used to reverse engineer their software. Additionally, PDB files are only generated by one compiler family (Microsoft) to our knowledge. Thus, solutions that depend on PDB files are not compiler-agnostic.

Microsoft has applied Vulcan to three different binary rewriting techniques: Control Flow Integrity (CFI) [Abadi et al., 2005], an extension to CFI called XFI [Erlingsson et al., 2006], and Software Memory Access Control (SMAC) [Abadi et al., 2009]. CFI determines the control flow graph (CFG) of the execution of a program, and then ensures via instrumentation that the program only follows the paths dictated by the CFG. CFI potentially mitigates a large number of low-level attacks, however its overhead is somewhat higher than other SFI techniques—about 16% on average. XFI expands on the work developed in CFI, dictating stricter properties for implementing a CFI system—whereas SMAC adds memory safety properties to CFI, allowing controlled modification of data sections that is uncircumventable due to CFI's underlying protections. XFI has higher overhead than CFI, with overheads between 5% and 100%, and adding SMAC to these projects adds more guard instructions.

5.2.3 Virtual Machines

The use of a large scale virtual machine such as VMWare [VMware, 2012] or Virtualbox [Oracle Corporation, 2012] tends to be impractical for fine-grained, intra-process, control-flow sandboxing because such VMs opaquely interpret untrusted code without attempting to infer its internal semantics. This limits them to traditional OS protections, such as process isolation. More fine-grained enforcement can lead to unacceptably high overheads. Smaller, more intelligent, per-process virtual machines can possibly achieve low overheads by interpreting blocks of instructions at a time. However, in general IRMs optimize VMs by in-lining the VM logic into the untrusted code and statically specializing it to the program via partial evaluation. This leads to much lower overheads and avoids security complications that tend to arise from self-modifying code.

More recently, the vx32 [Ford and Cox, 2008] system was developed, using a small virtual machine to perform sandboxing of a binary. The performance of vx32 is quite impressive for a virtual machine, but performs worse than a binary rewriter. Also, vx32 has only been implemented for Linux based operating systems and its safety has not been proved or machine-verified.

5.2.4 Restrictionless Binary Rewriting

There has been some work that focuses on arbitrary binary rewriting, however the results are narrowly scoped or short of production level.

The first of such rewriters that was developed is Etch [Romer et al., 1997], which allows for in place binary rewriting. In place rewriting only supports two kinds of code transforms: size-reducing and peep-hole. Size-reducing transforms replace an instruction sequence in-place with a smaller one, and pad the remainder with no-operation instructions. Peep-hole rewriting replaces an instruction with a call to a subroutine that implements a longer instruction sequence in its place. These transforms do not suffice to provably enforce control-flow safety, since many computed jump instructions (e.g., returns, which are only one byte long on x86) are too small to support either transformation technique. In addition, the peep-hole approach incurs high overhead due to the higher number of branches.

The Azure rewriting system [Yardimci and Franz, 2009] performs static rewriting in order to identify possible parrelization candidates to dynamically recompile at runtime to improve performance. Azure does not require any source code or metadata, but is currently only supported for the PowerPC architecture.

SecondWrite [Smithson et al., 2010] uses a disassembler and intermediary representation to attempt instrumentation combined with architectural independence. SecondWrite uses a rewriting system that requires the original code section to be retained and left executable, which allows for more attack vectors. Also, SecondWrite uses a jump table to handle indirect calls, effectively turning each indirect call into a sequence of conditional direct calls to new addresses. Effectively, the number of guard instructions is $2n$ where n is the number of permissible targets of each branch. Also, SecondWrite is currently unable to handle any mainstream binaries [Smithson et al., 2010]. The system also has a very large trusted computing base that currently consists of over 120,000 lines of C++ code. This makes it very hard to formally verify.

5.3 Transparency

Past work on IRM theory has defined IRM correctness in terms of soundness and transparency (i.e., behavioral equivalence) [Hamlen et al., 2006, Ligatti et al., 2005]. Transparency is defined in terms of a trace-equivalence relation that demands that equal program inputs (including non-deterministic choices) yield equivalent traces of observable program actions. Traces are equivalent if they are equal after erasure of irrelevant events (e.g., stutter steps).

Chudnov and Naumann provide the first formal proof of transparency for a real IRM [Chudnov and Naumann, 2010]. The IRM enforces information flow properties, so transparency is there defined in terms of program input-output pairs. In lieu of machine-certification, a written proof establishes that all programs yielded by that particular rewriting algorithm are transparent. The proof is therefore specific to one rewriting algorithm and does not necessarily generalize to other IRM systems.

CHAPTER 6
CONCLUSION

This dissertation presented a binary rewriting framework for the x86 architecture that requires no code-producer cooperation and causes minimal overhead.

Chapter 2 presented the issues inherent in static x86 disassembly. Three different novel approaches for handling disassembly were presented. Shingled disassembly, presented in Section 2.3, provides full execution paths through a sequence of bytes regardless of instruction aliasing, which is necessary for preserving semantic information when binary rewriting. The PPM Disassembler presented in Section 2.4 is the first work on machine learning-based disassembly, achieving improved accuracy over IDA Pro using a kth-order Markov model. Finally, the FSM disassembler presented in Section 2.5 presents a more efficient linear disassembly algorithm that shows improved runtimes and disassembly accuracy over IDA Pro.

In Chapter 3, the binary rewriting framework was presented. We describe how the problem of static x86 disassembly undecidability can be circumvented by using a shingled disassembly to preserve all semantic information. Additionally, we describe a solution to computed jump preservation using a dynamic lookup table.

Two binary rewriting systems were created to extend this framework to solve real-world problems. STIR extended these techniques in Section 3.2 to randomize a binary at load-time, protecting against ROP attacks with high probability. We used this to automatically transform hundreds of Windows and Linux binaries into self-randomizing programs. Each time the transformed programs load, they re-randomize themselves at the basic block level. This protects them against return-oriented programming attacks and other attacks that hijack control flows. STIR binaries showed an overhead of just 1.4% on average.

REINS then extended these techniques in Section 3.3 to enforce machine verifiable security constraints. Using the Software Fault Isolation (SFI) approach developed by PittSFIeld [McCamant and Morrisett, 2006], our rewriter REINS in-lines guard instructions, that restrict flows at runtime to policy-permitted targets. The safety of the resulting code is verifiable using a straight-line disassembler we developed in just 1500 lines of OCaml code. With this enforcement in place, we were able to develop a policy language in order to specify application-specific security policies based on system call interposition. With all of these precautions in place, REINS binaries showed an overhead of only 2.5%.

In Chapter 4, we present the groundwork for statically proving the transparency of a binary rewriting system. We implemented a proof of concept for this system in ActionScript byte code, and went on to discuss the difficulties inherent in proving transparency for the x86 architecture.

Finally in Chapter 5 we discuss associated work in the area.

We consider these systems to be a significant contribution to the field since all previous x86 rewriting frameworks require code-producer cooperation [Yee et al., 2009], are unable to handle many non-trivial rewriting policies [Romer et al., 1997], or introduce significant attack vectors [Smithson et al., 2010]. This is a necessary evolution in binary rewriting technology that secures a much larger class of x86 COTS binaries than was possible previously.

The prototypes developed as part of this dissertation are proofs-of-concept, not production-level systems. In order to keep the projects manageable for a very limited development staff (*viz.* two students and one professor), the prototype intentionally omits numerous features whose design and implementation should be the subject of future work:

1. Extending our system with a simple-to-use API that allows users to write their own rewriter policies is necessary to better disseminate our rewriting framework for others to use. Currently the rewriting rules for REINS and STIR are hardcoded into separate

binary rewriters for each system. Future dissemination of our framework depends on first modularizing the framework for component-wise use.

2. At the time of publication, both Reins and Stir use IDA Pro [Hex-Rays, 2012] as their means of execution path pruning. Since IDA Pro is expensive and prone to mistakes, switching over to using our FSM Disassembler detailed in Section 2.5 would remove any reliance on expensive outside software and also decrease the number of disassembly errors that may have to be manually fixed.

3. Many mainstream Windows binaries such as Microsoft Office heavily rely on Windows COM Object libraries. Due to the nature of COM objects, which implement and exchange interfaces composed of dynamically generated code pointer tables, Reins is currently unable to support them. A wrapper function for COM object creation would have to be created for each object type, such that code pointers could be dynamically updated. This is a significant engineering task, but once accomplished, a large number of additional binaries should be rewritable via Reins.

4. Currently Reins is able to verify complete mediation for rewritten binaries, but there is no guarantee that the mediator implements any particular high-level policy. For simple security policies, this task is trivial by hand, but in order to minimize the TCB, we envision the development of certifying compilation technology for mediators.

5. As discussed in Section 4, currently we can statically prove transparency for ActionScript bytecode but not for the x86 architecture. There are significant hurdles with x86 due to the inclusion of computed jumps. Discovery and innovation of transparency verification technologies for native code IRMs is important for assuring customers that the security system will not adversely affect well-behaved software—a common concern for many administrators and users.

REFERENCES

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1), 2009.

Adobe Systems Incorporated. ActionScript virtual machine 2 (AVM2) overview, May 2007.

Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1986.

Starr Andersen. Part 3: Memory protection technologies. In Vincent Abella, editor, *Changes in Functionality in Windows XP Service Pack 2*. Microsoft TechNet, 2004. `http://technet.microsoft.com/en-us/library/bb457155.aspx`.

Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. CodeSurfer/x86—a platform for analyzing x86 executables. In Rastislav Bodik, editor, *Proceedings of the 14th International Conference on Compiler Construction (CC)*, pages 250–254, 2005.

Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 255–270, 2005.

Andrej Bratko, Bogdan Filipič, Gordon V. Cormack, Thomas R. Lynam, and Blaž Zupan. Spam filtering using statistical data compression models. *Journal of Machine Learning Research*, 7:2673–2698, 2006.

Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 27–38, 2008.

Business Software Alliance. Software industry facts and figures, 2010. `http://www.bsa.org/country/public%20policy/~/media/files/policy/security/general/sw_factsfigures.ashx`.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 559–572, 2010.

Andrey Chudnov and David A. Naumann. Information flow monitor inlining. In *Proceedings of the 23rd Computer Security Foundations Symposium (CSF)*, pages 200–214, 2010.

John G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–75, 1997.

John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

Corelan Team. Mona, 2012. `redmine.corelan.be/projects/mona`.

G. V. Cormack and R. N. S. Horspool. Data compression using dynamic Markov modeling. *The Computer Journal*, 30(6):541–550, 1987.

Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

Mark Dowd. *Application-Specific Attacks: Leveraging the ActionScript Virtual Machine*. IBM Global Technology Services, April 2008.

Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, Inc., San Francisco, California, 2008.

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

Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

Alan Eustace and Amitabh Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 303–314, 1995.

FireEye. Heap spraying with ActionScript, 2009. `http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html`.

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

Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 293–306, 2008.

Gartner. Gartner says worldwide enterprise software revenue to grow 9.5 percent in 2011. `http://www.gartner.com/it/page.jsp?id=1728615`, June 2010.

GNU Project. Gnu binary utilities. `http://sourceware.org/binutils/docs-2.22/binutils/index.html`, 2012.

Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

Rajiv Gupta. Generalized dominators and post-dominators. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 246–257, 1992.

Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):175–205, 2006.

Kevin W. Hamlen, Vishwath Mohan, and Richard Wartell. Reining in Windows API abuses with in-lined reference monitors. Technical Report UTDCS-18-10, The University of Texas at Dallas, 2010.

Kevin W. Hamlen, Micah M. Jones, and Meera Sridhar. Aspect-oriented runtime monitor certification. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 126–140, 2012.

Hex-Rays. The IDA Pro disassembler and debugger, 2012. `www.hex-rays.com/idapro`.

Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, pages 571–585, 2012.

Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*, chapter 4: The Age-Old Art of Hooking, pages 73–74. Pearson Education, Inc., 2006.

INRIA. The Coq proof assistant. `http://coq.inria.fr`, 2012.

Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual. `http://download.intel.com/products/processor/manual/325462.pdf`, August 2012.

Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20(1):503–581, 1994.

Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14 (3):339–395, 1992.

Micah Jones and Kevin W. Hamlen. Disambiguating aspect-oriented security policies. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 193–204, 2010.

Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, 2012.

Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 267–282, 2012.

Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 423–427, 2008.

Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 43–50, 2010.

Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 214–228, 2009.

Nithya Krishnamoorthy, Saumya Debray, and Keith Fligg. Static detection of disassembly errors. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, pages 259–268, 2009.

Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004.

David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM (CACM)*, 52(7):107–115, 2009.

Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, pages 355–373, 2005.

Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, pages 209–224, 2006.

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

Mitre Corporation. CVE-2010-2216, 2010a. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2216`.

Mitre Corporation. CVE-2010-2215, 2010b. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2215`.

Alistair Moffat and Andrew Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

Oracle Corporation. Position-independent code. In *Linker and Libraries Guide*. 2010. `http://docs.oracle.com/cd/E19082-01/819-0690/chapter4-29405/index.html`.

Oracle Corporation. VirtualBox. `http://www.virtualbox.org`, 2012.

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, pages 601–615, 2012.

PaX Team. PaX address space layout randomization (ASLR), 2003. `http://pax.grsecurity.net/docs/aslr.txt`.

Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, pages 60–69, 2009.

Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, 1997.

Jonathan Salwan. ROPgadget, 2012. `http://shell-storm.org/project/ROPgadget`.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the Workshop on Binary Translation (WBT)*, 2001.

Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 45–54, 2002.

Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, 2004.

Leon Shapiro and Ehud Y. Sterling. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1994.

Christopher Small and Margo I. Seltzer. A comparison of OS extension technologies. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 41–54, 1996.

Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. Binary rewriting without relocation information. Technical report, University of Maryland, November 2010.

Solar Designer. “return-to-libc” attack. Bugtraq, August 1997.

Meera Sridhar and Kevin W. Hamlen. Model-checking in-lined reference monitors. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 312–327, 2010.

Meera Sridhar, Richard Wartell, and Kevin W. Hamlen. Hippocratic binary instrumentation: First do no harm. Unpublished manuscript, to be submitted, 2012.

Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, 2005.

W. J. Teahan. Text classification and segmentation using minimum cross-entropy. In Joseph Mariani and Donna Harman, editors, *Proceedings of the 6th International Conference on Recherche d’Information et ses Applications (RIAO)*, pages 943–961, 2000.

W. J. Teahan, Rodger McNab, Yingying Wen, and Ian H. Witten. A compression-based algorithm for Chinese word segmentation. *Journal of Computational Linguistics*, 26(3): 375–393, 2000.

Arjan van de Ven. New security enhancements in Red Hat Enterprise Linux v.3, update 3. Whitepaper WHP0006US, Red Hat, 2004. `http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf`.

VMware. Desktop virtualization. `http://www.vmware.com`, 2012.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, volume 3, pages 522–536, 2011.

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, December 2012a.

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, 2012b.

Efe Yardimci and Michael Franz. Mostly static program partitioning of binary executables. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(5), 2009.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, pages 79–93, 2009.

VITA

Richard Wartell was born in Eureka, California on May 4th, 1986, the son of Michael and Ruth Wartell. When he was three, his father was quizzing his eight year old brother Justin on math and he began answering the questions faster than his brother. This lead to him constantly being ahead in Mathematics by about two years of schooling. Then in middle school, due to the structure of math classes, he was able to advance further such that by the second half of 7th grade he was taking Honors Calculus at Indiana University Purdue University Fort Wayne (IPFW).

In the meantime, Richard had developed an intense love of all things computers, digging in to the guts of his home PC with family friend Pam Olson and scaring his father to no end. This lead to the pursuit of 3D modelling and animation courses at IPFW, as well as introductory Computer Science classes. By the middle of his junior year of high school, it became apparent that if he worked hard, he would be able to pursue a Bachelors in Mathematics from IPFW by the time he graduated from high school, and he did just that.

After graduating, he finished a Computer Science degree at Purdue in three years. During his final year at Purdue, he was contacted by The University of Texas at Dallas and offered the Johnson Scholarship, a full ride for two years in their Ph.D. program. He accepted their offer, and following a Summer internship at Ball Aerospace in Boulder Colorado, he pursued his Ph.D. in C.S. at UT Dallas working under Dr. Kevin W. Hamlen.

After finishing his Ph.D. Richard accepted a position with Mandiant as a Malware Analyst and Incidence Response Security Consultant.