ABSTRACT, SAFE, TIMELY, AND EFFICIENT BINARY MODIFICATION

by

Andrew R. Bernat

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 4/25/2012

The dissertation is approved by the following members of the Final Oral Committee:

To Jenn, Danny, and Becky.

Acknowledgments

During the course of my graduate career, I have been fortunate to receive advice, support, and encouragement from many people.

My adviser, Bart Miller, taught me how to do research; how to effectively present ideas, whether individually or to an audience; and how to take the time to do things right. I also owe to him an interest in woodworking, where many of the same truths hold. I can only hope that his near-infinite patience will be rewarded.

The other members of my committee, Jeff Hollingsworth, Tom Reps, Michael Swift, and Paul Wilson, through their thoughtful comments and suggestions, helped make this dissertation what it is. Thank you to Somesh Jha and Charlie Fischer for assisting with my preliminary exam. Though the ideas discussed in this work have diverged from the ideas I presented there, their input has guided and shaped my research.

I have been fortunate to be the member of a dynamic and flourishing research group, the Paradyn Project. There are far too many members in my tenure to list individually. Special thanks to Ari, Phil, Vic, and Karen, for welcoming a novice programmer to Madison; to Kevin, for sharing the fascinating and difficult area of malware analysis and instrumentation; to Nate, for always answering questions graciously and patiently; to Bill, for extensive design discussions and help on my notation; to Laune, for showing that symbol tables (and, by extension, binaries) cannot be trusted; and last, but not least, to Brian, Matt, and Madhavi, for allowing me to focus on research instead of the day-to-day details of implementation.

During my years in Madison I have been fortunate to have made lifelong friends. In a, by no means complete, list: Hao Wang, Jaime Frey, and Sharon and Brad Kahn have helped this town become my home. The Madison Jewish community, including Temple Beth El, the Beth Israel Center, and Hillel, welcomed me and my family and gave us a spiritual home.

Will Benton provided the template in which I have written this document, and recommends that I thank the Academy.

My college professors nurtured in me a fascination with computing. Particular thanks to Ran Libeskind-Hadas for teaching me that learning can be fun, and Geoff Kuenning for introducing me to the fascinating area of current systems research.

My family provided constant support. In particular, my parents, Andrew and Valerie Bernat, and my sister, Robin Latour, provided never-ending encouragement and faith in my abilities. I hope someday that my father is mistaken for me, rather than the other way around. My grandfather, Julian Bernat, is my role model in every aspect of my life. Jenn's parents, Vik and Sue Hands-Renwick, and her brother Tim, welcomed me into their family and their hearts. I owe much to their patience and love.

Finally, to my wife and love of my life, Jenn Bernat, who made this possible. She walked beside me, supported me, and carried me during this long journey.

Contents

List of Tables

List of Figures

ABSTRACT, SAFE, TIMELY, AND EFFICIENT BINARY MODIFICATION

Andrew R. Bernat

Under the supervision of Professor Barton P. Miller
At the University of Wisconsin-Madison

Program binaries are commonly held to be an execute-only program form: rigid, lacking in clear structure, complex to extend and difficult to modify. However, there are several benefits to be gained from modifying binaries rather than another program form: the effects of the compiler upon the program are clearly present; binary modification does not require access to source code, which may be unavailable; and users may manipulate programs while they execute, which is impossible with other forms of program modification.

In this dissertation, we develop and refine four desired properties of a binary modification toolkit: abstraction, safety, timeliness, and efficiency. By abstraction, we mean that a user should operate in terms of familiar structural representations, such as functions, loops, or basic blocks, instead of directly on instructions. By safety, we mean that modification should preserve the visible behavior of code that was not explicitly modified and the structural validity of the binary as a whole. By timeliness, we mean that a toolkit should allow modification of a binary at any time in its execution continuum, from a file on disk to actively executing code. By efficiency, we mean that modification should impose cost that is both low and proportional to the amount of modified code and the frequency with which it is executed.

We then describe three techniques that allow us to achieve these properties. First, we demonstrate that the CFG, an abstraction that represents the binary program's structure, can also be used to modify this structure and thus the binary as a whole. By leveraging the CFG, we allow users to operate in terms of familiar and natural constructs rather than requiring them to understand the idiosyncrasies of particular instruction sets. Second, we further refine techniques for code replacement, allowing us to modify a program binary at any time in its execution continuum while preserving proportional cost. Third, we present a technique based on a formal understanding of the characteristics of binary code that allows us to modify the structure of the binary without changing its user-visible behavior, even when the binary attempts to detect such modifications.

Barton P. Miller

Abstract

Program binaries are commonly held to be an execute-only program form: rigid, lacking in clear structure, complex to extend and difficult to modify. However, there are several benefits to be gained from modifying binaries rather than another program form: the effects of the compiler upon the program are clearly present; binary modification does not require access to source code, which may be unavailable; and users may manipulate programs while they execute, which is impossible with other forms of program modification.

In this dissertation, we develop and refine four desired properties of a binary modification toolkit: abstraction, safety, timeliness, and efficiency. By abstraction, we mean that a user should operate in terms of familiar structural representations, such as functions, loops, or basic blocks, instead of directly on instructions. By safety, we mean that modification should preserve the visible behavior of code that was not explicitly modified and the structural validity of the binary as a whole. By timeliness, we mean that a toolkit should allow modification of a binary at any time in its execution continuum, from a file on disk to actively executing code. By efficiency, we mean that modification should impose cost that is both low and proportional to the amount of modified code and the frequency with which it is executed.

We then describe three techniques that allow us to achieve these properties. First, we demonstrate that the CFG, an abstraction that represents the binary program's structure, can also be used to modify this structure and thus the binary as a whole. By leveraging the CFG, we allow users to operate in terms of familiar and natural constructs rather than requiring them to understand the idiosyncrasies of particular instruction sets. Second, we further refine techniques for code replacement, allowing us to modify a program binary at any time in its execution continuum while preserving proportional cost. Third, we present a technique based on a formal understanding of the characteristics of binary code that allows us to modify the structure of the binary without changing its user-visible behavior, even when the binary attempts to detect such modifications.

1 Introduction

Program binaries are commonly held to be an execute-only program form: rigid, lacking in clear structure, complex to extend and difficult to modify. This is not the case. With the proper techniques, derived from a formal understanding of the characteristics of program binaries, a binary can be extended and modified up to and during execution of the program. In turn, binary modification presents several benefits when compared to source-level or compile-time modification. Binary modification does not require source code, debugging information, or other information that is not directly required to execute the program. In the commercial or security domains, the binary is frequently the only available program form; even in other domains, binary modification may be required due to a lack of source code, binary-only vendor-provided libraries, or the need to include the effects of compilation and linking on a program.

Binary modification has become a fundamental enabling technology for a wide range of domains, including optimization [10, 70], performance analysis [43], dynamic ("hot") patching [40], testing [78], cyberforensics [45, 74], program auditing [21, 79], precise behavior monitoring [21], and attack detection [51]. Binary modification can either be applied to a binary on disk (*binary rewriting*) or to an executing process (*dynamic instrumentation*). The earliest uses of binary modification focused on performance analysis and whole-program optimization. Performance tools used dynamic search techniques to automatically find program bottlenecks [28] and inserted tracing code to measure basic block execution paths [7]. Optimization tools inlined calls across object file boundaries [70] or rearranged executing programs to improve cache performance [5].

Our work distinguishes itself from other research into binary modification toolkits in four significant ways: *abstraction*, *timeliness*, *safety*, and *efficiency*. For abstraction, we use a high-level model of the binary based on its control flow graph (CFG), and users manipulate this model instead of operating directly on instructions or other low-level representations. Such a high-level model greatly reduces the knowledge of program or platform idiosyncrasies previously needed to modify programs. By timeliness, we mean that we allow users to modify a binary at any time in the *execution continuum*: pre-execution, before the code selected to be modified has executed for the first time, or while such code is actively executing, and ensure these modifications take effect immediately. For safety, we use a formal model of *structural validity* that ensures that modified binaries do not contain illegal control flow and *compatible visible behavior* that ensures that the behavior of code that was not explicitly modified is preserved. For efficiency, we require that our cost is proportional to the amount of modification or instrumentation and the frequency with which it is executed; as a result, we impose overhead (both in space and execution time) that is the same or lower than other binary modification toolkits while providing cleaner interfaces and stronger correctness guarantees.

This dissertation describes the design and implementation of several novel techniques that we use to accomplish these four goals. First, we define *structured binary editing*, a binary modification technique that ensures the modified binary is structurally valid. This technique allows users to modify the binary by transforming the familiar representation of its control flow graph (CFG) with an algebra of valid CFG transformations. Manipulating the CFG instead of the binary directly removes the need for an instruction-level understanding of the binary while ensuring that, by maintaining a valid CFG, the user does not accidentally create an invalid binary. Second, we define *anywhere binary modification*, an approach that applies conventional definitions of functions, loops, and basic blocks to binaries whose code does not naturally support these structures due to optimization or obfuscation. Third, we define *anytime code replacement*, a technique that can replace original code with modified code at any point in the execution continuum while imposing cost proportional to these changes. Fourth, we define *sensitivity resistant code relocation*, a technique that allows us to incorporate new code into a binary while preserving the visible behavior of the original code.

1.1 Guiding Principles

In this section, we elaborate on our guiding principles of abstraction, safety, timeliness, and efficiency.

Abstraction Binaries are complex mixtures of code and data; we seek to hide these complexities and the idiosyncrasies of binary modification within a set of familiar, high-level abstractions. We leverage the control flow graph, which has a well understood hierarchy of abstractions, such as basic blocks, control flow edges, loops, and functions, that represent program structure. In particular, we wish to do three things. First, to eliminate the need for users to have detailed knowledge of instruction sets and operating system interfaces and reduce the difficulty of porting tools between different architectures and operating systems by encapsulating the differences between these platforms. Second, to represent higher level program behaviors, such as "entry to a function", that may not be obvious at the instruction level[1]. Third, to hide many of the complexities of binaries, particularly programs that have been heavily optimized or obfuscated. For example, functions and blocks may overlap and share code bytes. Modeling these these structures as independent abstractions hides this sharing from the user when such information is not needed, and exposes it when desired by the user.

Safety It is important for binary modification toolkits to both preserve the binary's structural validity and ensure that the toolkit itself does not

[1]While entry to a function may *seem* like an obvious behavior associated with executing a `call` instruction, it is in practice more complex particularly for highly optimized or obfuscated code, where call instructions are used to access the program counter and jumps often replace calls to optimize call/return behavior.

alter the visible behavior of the binary. A binary is structurally valid if it has valid control flow and only executes valid instructions; for example, it must not branch to data (not including code in data), unallocated memory, or the middle of an instruction. Toolkits often further modify the binary, such as to insert additional infrastructure code; we these modifications must not change the visible behavior of the program. We define visible behavior in terms of denotational semantics (informally, I/O behavior), which allows internal details of execution to change so long as a binary's output is the same. Current binary modification techniques require the user to ensure structural validity and rely on ad-hoc techniques to preserve visible behavior. We believe that these approaches are not sufficient; instead, we rely on strong formal models of both structural validity and visible behavior to guarantee the safety of our techniques.

Timeliness Users may wish to modify a piece of code at any time in the *execution continuum*: pre-program execution, while the selected code is not executing (e.g., at the start of program execution), or while the selected code is currently executing. This principle presents an interesting challenge: we must present a single modification and instrumentation interface that can be efficiently supported by a tool infrastructure that provides both binary rewriting and dynamic instrumentation. In addition, we believe that any such modification should take effect immediately rather than waiting for the selected code to be executed again (e.g., a different loop iteration or function call). For example, the SD-Dyninst research prototype [60] relies on the ability to instrument the current instruction and have this instrumentation take immediate effect; this is a critical component of its ability to analyze and instrument malware.

Efficiency Modifying and instrumenting a binary imposes overhead, both in space (the size of the modified binary) and execution time. We believe this overhead should be proportional to the amount of modified code and the frequency with which it is executed. Inversely, code that has not been explicitly modified should have no associated overhead. Finally, if a user removes a modification (e.g., removing inserted code) the costs associated with the modification should similarly be eliminated.

1.2 Structured Binary Editing

The first contribution of this dissertation is structured binary editing, a technique for modifying a program binary by manipulating its control flow graph (CFG). The CFG describes the structure and possible execution paths of a binary, and we demonstrate that it can be used to manipulate these features as well. Structured binary editing is conceptually simple: the user manipulates the CFG to describe their desired resulting binary, including inserting new code, copying existing code, modifying existing control flow paths, or removing undesired code. This technique can also serve as the basis for more complex modifications, such as modifying the data structures of the binary [49]; however, such modifications are beyond the scope of this work. We then use this modified CFG to generate a new binary. In addition to allowing users to manipulate the structure of the binary at a high level, this approach allows us to guarantee the resulting binary is structurally valid by ensuring the CFG itself is valid. Structured binary editing consists of four components: formalizing structural and CFG validity, ensuring that the transformed CFG is valid by defining a set of valid and useful CFG transformations, handling the insertion of new code into a valid CFG, and ensuring that modification does not invalidate calculated, or *indirect*, control flow.

First, we formalize structural validity of binaries and define CFG validity. We use a simple model of structural validity: a binary is structurally valid if it has legal control flow and thus will only execute valid instructions. A binary that attempts to branch to an unallocated address, into the middle of an instruction, or execute actual data (but not code in data) is structurally invalid. We define a CFG to be valid if it represents a structurally valid binary. For example, a basic block with no out-edges that is not an exit block is invalid, since such a block does not represent actual code.

Second, we define a modification algebra of graph transformations that is closed under CFG validity. A graph transformation replaces a particular subgraph of an input graph G with a pre-defined replacement graph [3, 26]. These transformations are purposefully simple and localized to avoid unexpected side-effects on program behavior. However, since our algebra is closed, transformations can be arbitrarily composed to perform complex modifications of the program.

Third, we address the insertion of new code into a valid CFG. We must ensure this new code does not introduce control flow of which we are not aware to ensure our CFG remains correct. We do this by defining *insertion transformations*, which insert a specific code sequence that matches a single conceptual operation, such as inserting a conditional branch or call, and has no other side-effects on the CFG. In addition, we allow users to insert *code snippets*, which are single-entry, single-exit regions of code. Once inserted, a snippet can be transformed as part of the original CFG, allowing users to construct more complex code sequences.

Fourth, we address indirect control flow. We first present our algebra of transformations in a simplified model that assumes that the control flow of the program is not affected by its data flow. In this simple model, a particular application of a transformation will have no control flow side-effects outside of the transformed code. Since transforming control flow frequently affects the data flow of the program by changing which instructions execute, this simplified model assumes that any changed data flow has no effect on control flow, such as by altering the target of an indirect branch. As a result, we can determine whether a transformation is valid or not for any possible application of that transformation.

Since indirect control flow uses values calculated at run-time, the data flow side-effects of a transformation can alter control flow outside of the transformed portion of the CFG. These *non-local* control flow changes in turn could cause an invalid CFG; for example, by corrupting a function pointer to target non-code. We address this problem by defining a slicing-based dataflow analysis that determines if the destinations of an indirect control flow instruction may be invalidated by a transformation of the CFG. We can either invalidate such transformations or insert runtime monitoring code at the option of the user.

For simplicity of description, we assume that a complete CFG can be derived from the input binary. This is not always possible to do with a static analysis due to the presence of exceptions, callbacks, indirect control flow, or self-modifying code. This problem can be overcome by augmenting static parsing with dynamic parsing techniques to identify a complete CFG during execution [60]. Such a CFG can then be modified and used to generate modified code; if the CFG is further developed by additional dynamic parsing, then the process can be repeated.

We define our CFG model and describe the concepts and implementation details of structured binary editing in Chapter 3, and describe a *CFG view* technique that allows us to apply structured binary editing to binary instrumentation in Chapter 5.

1.3 Projections on a CFG: Functions and Loops

The second contribution of this dissertation extends structured binary editing to operate on functions and loops, abstractions that provide a familiar interface to the binary. For example, a user can insert new code at a function entry or at the top of a loop instead of having to operate at the instruction or basic block level. Such an approach provides significant value by encapsulating binary complexity, but also poses significant design challenges. Modern optimizing compilers focus on efficiency of generated code instead of generating simply structured code; as a result, the mapping of these abstract representations onto the binary may be complex.

Previous approaches [11, 36, 38, 41, 52, 57, 68, 69] use an overly simple mapping of these functions and loops to instructions, including mapping the entry of a function or loop to its first instruction, assuming that functions do not share code (a common optimization), and assuming functions only end at return instructions instead of optimized return sequences. Consider the example shown in Figure 1.1 of a function whose preamble code (stack setup) was omitted by the compiler. Since previous approaches map the entry of a function to its first instruction, they would generate the instrumented CFG shown in Figure 1.2a. This CFG is incorrect, as entry instrumentation may

```
int main(int argc) {
  do {
    argc -= 1;
  } while (argc > 0);
  return argc;
}
```

(a) Code Listing

```
400450: sub    $0x1, %edi
400453: test   %edi, %edi
400455: jg     400450
400457: mov    %edi, %eax
400459: ret
```

(b) Disassembly

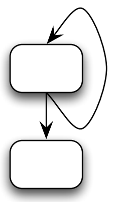

(c) Function CFG

Figure 1.1: Code listing, disassembly, and CFG for an example function whose first instruction executes multiple times per function invocation. This example was generated by GNU GCC version 4.1.2 at optimization level O2.

execute multiple times per function invocation.

We describe more sophisticated abstractions of functions and loops that encapsulate the complexity present in optimized or obfuscated binaries. We extract these abstractions from the interprocedural CFG and map them back to the CFG instead of directly to the underlying binary. Instrumentation and modification of these abstractions is applied to the underlying CFG using the transformation algebra described in Section 1.2. For example, entry instrumentation is performed by creating a distinct entry block, as shown in Figure 1.2b; this allows correct instrumentation even of optimized functions.

For several reasons, the functions and loops we identify in the binary may not precisely correspond with functions and loops in the source code. For example, inlined functions will not be distinguished from their callers, and unrolled loops may not be identified. Instead, we attempt to identify structures with familiar characteristics to source code functions and loops that also include the effects of compilation.

We define our function and loop abstractions and discuss modification of

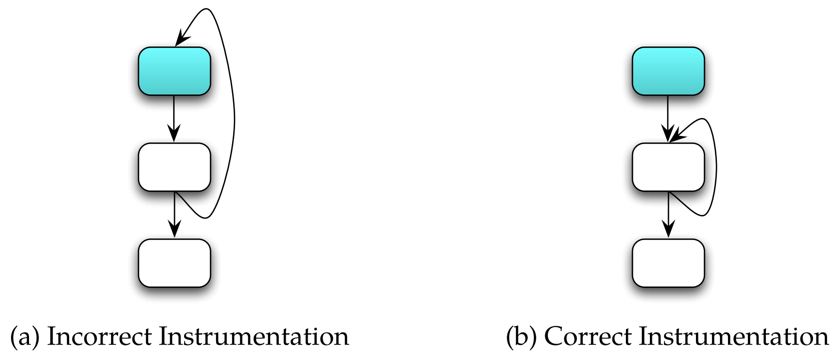


Figure 1.2: Examples of instrumenting the function shown in Figure 1.1, with instrumentation shown shaded. Figure (a) shows the result of prior approaches that instrument a function by instrumenting its first instruction; as a result, instrumentation may execute multiple times per function invocation. Figure (b) shows our approach, which creates a new entry block and inserts instrumentation there without disturbing the rest of the function.

these abstractions in Chapter 4.

1.4 Anytime Code Replacement

The third contribution of this dissertation is a technique for replacing the code in a binary at any time while imposing proportional overhead; we call this technique *anytime code replacement* [8]. The techniques described in Sections 1.2 and 1.3 allow a user to modify the CFG. To apply these modifications to the binary we first use the CFG to generate new binary code, and then replace the corresponding regions of original code with this new code. Due to the presence of data in code and indirect control flow, it is often difficult to replace the original code in place. Instead, we append the new code to the binary and *patch* the original code with *interception branches* to the new code. Patch-based code replacement has been used by several binary modification toolkits [11, 36, 38, 52, 68]; however, these toolkits cannot patch executing code and incur unnecessary overhead due to frequent execution of interception branches.

We define executing code to include both the current point of execution (or points for multithreaded programs) as defined by the program counter as well as code referenced in memory, such as return addresses on the stack or function pointers. We must ensure that execution immediately moves to the replacement code instead of continuing in the original code. If not, we cannot guarantee that modification of the binary will take effect immediately, or even in the near future in the case of a long-running loop. We address this problem with *state interception*, which operates by directly modifying process state to move execution from the current execution point to the corresponding destination in the modified code.

Surprisingly, frequent execution of branches, even unconditional branches, results in significant execution overhead. Our experiments show that this is caused by branching between two separate regions of code, which increases the pressure on the instruction cache, and by branches causing bubbles in the processor pipeline, which stalls instruction fetch and decode. Due to these factors, we minimize the frequency of executing interception branches with *region patching*. Instead of copying single instructions, we identify a *region* that includes all modified areas of the CFG. This region may vary in size from a single basic block to a set of functions. We then consider this entire region to be modified and patch only the entry points to this region with interception branches. This technique is a generalization of the function patching approach previously used by several binary modification toolkits [11, 36, 38, 68], and greatly decreases the frequency of executing interception branches.

As with structured binary editing, our description of anytime code replacement assumes the existence of a complete CFG. If the CFG is incomplete, we may either augment it with dynamic analysis or mark the incomplete regions of the CFG unmodifiable. Dynamic analysis will result in a complete CFG, but must be performed at runtime; in contrast, marking regions of the CFG as unmodifiable will allow the user to rewrite the remaining portions of the binary.

We describe region patching and state interception in Chapter 6.

1.5 Sensitivity-Resistant Code Relocation

The fourth contribution of this dissertation is a technique for replacing code in a binary while preserving the behavior of both the original code and the surrounding code. Such replacement frequently changes the contents of registers and memory as a side effect. For example, region patching overwrites original code with interception branches, which alters the memory corresponding with this code; adds new code, which allocates additional memory; and executes new code, which will perceive a different program counter value. Other code replacement techniques may also overwrite original code, allocate additional memory, and execute new code at a different address. These changes alter the contents of registers, such as the program counter, and memory, and may in turn alter the behavior of instructions that use these changed registers or memory as inputs. For example, a moved call instruction will produce a changed return address. We call these instructions *sensitive* to code replacement.

Binary modification toolkits seek to compensate for the altered behavior of sensitive instructions using *code relocation*, a technique that produces code that has compatible visible behavior with the original code. Previous approaches have relied on ad-hoc definitions of sensitivity and visible behavior and thus impose unnecessary overhead [10, 41, 50] or may fail to preserve compatible behavior [11, 36, 38, 52, 57, 68, 69]. We describe *sensitivity-resistant code relocation*, a technique that relies on a formal specification of sensitivity to both preserve correct behavior while often imposing lower overhead than previous approaches [9]. This technique consists of four components: a model of instruction sensitivity, a formalization of the compatible visible program behavior that we wish to preserve, an analysis for identifying *externally* sensitive instructions that will alter this behavior, and an efficient compensation technique for handling such instructions.

First, we describe a model of instruction sensitivity. We classify the effects of code replacement on the program and identify four classes of affected instructions: instructions in modified code that perceive a different value for the PC, instructions that access original code that was overwritten, instructions that access newly allocated memory, and instructions whose successors were moved. For each category we define how the inputs and outputs of the instruction are changed by code replacement.

Second, we formalize compatible visible behavior in terms of denotational semantic equivalence. When we modify the program, we maintain a relationship between the CFG and the modified binary. We formalize *output flow compatibility*, a stricter approximation of denotational semantic equivalence that leverages this relationship to make the determination of equivalence tractable.

Third, we describe a dataflow analysis for identifying externally sensitive instructions. Not all sensitive instructions are externally sensitive. Consider a moved call instruction; since executing a call records a PC-dependent return address, a call is PC-sensitive. However, if this return address is only used for control flow, such as by a return instruction, then moving the call will have no effect on the program's behavior; in this case the call would be *internally* sensitive. If the return address is used as part of a pointer calculation, such as for a position-independent jump table; then modifying its value may alter the program's behavior and thus the call would be externally sensitive. Our external sensitivity analysis uses program slicing [33, 56] to determine which instructions may be affected by a sensitive instruction, and symbolic evaluation [16] to determine if these effects would break output flow compatibility; if so, we conclude the instruction is externally sensitive.

Fourth, we define a framework for low-overhead compensation of externally sensitive instructions. Relocation typically replaces each such instruction with a sequence that emulates the original instruction's behavior. This approach is straightforward but may miss opportunities for greater efficiency. We describe *group transformations* that replace a sequence of affected code as a single unit; our experiments show this technique results in a 23% decrease in overhead when instrumenting position-independent code.

Our sensitivity model only addresses the effects of moving, adding, and overwriting code. Inserting code into the binary may affect its behavior in other ways, such as increasing the execution time, modifying library state, modifying operating system state, or changing the layout of the heap by allocating memory. Although we do not address these effects on the program in this work, we believe our approach could be extended to do so; however, such an extension is beyond the scope of this work.

We describe these four components in Chapter 7.

2 Related Work

This dissertation explores binary modification and instrumentation. In this chapter, we survey related work that is closely related to our research. We begin by describing existing research in binary modification related to structured binary editing (Section 2.1). Next, we discuss related techniques for modifying binaries in terms of functions and loops (Section 2.2). We then relate anytime code replacement to existing binary code replacement techniques: patch-based, just-in-time (JIT), and in-place code replacement (Section 2.3). Next, we discuss other code relocation techniques that are similar to sensitivity-resistant code relocation (Section 2.4). We conclude the chapter with a summary of the related work (Section 2.5).

We implemented our new techniques and approaches in the Dyninst binary modification toolkit [11], replacing the techniques used previously. We compare our approach to these previous techniques in addition to the techniques used by other binary modification toolkits. When we refer to Dyninst, we specifically refer to Dyninst 7.0.

2.1 Binary Modification Interfaces

A binary modification toolkit provides an interface that allows a user to modify a binary program, either on disk or during execution. In this section, we discuss the interfaces used by current binary modification toolkits. We classify these interfaces by the abstractions that the user is given to manipulate: instructions, control flow graphs (CFGs), or functions. Our approach operates on the CFG and builds upon previous work in binary CFG modification as well as the related area of compile-time CFG modification.

We do not discuss toolkits that only provide binary instrumentation. Instrumentation inserts new code into the binary with the intent of leaving the original binary's behavior unmodified; thus, instrumentation is a subset of modification. While it is possible to insert instrumentation that alters the behavior of a program, such as code that branches from instrumentation to a different location in the program, doing so requires significant expertise by the user; binary modification toolkits attempt to make this expertise unnecessary. We discuss instrumentation toolkits where the technical details are relevant in the later sections of this chapter.

Instruction Modification Interfaces

The EEL [36], Etch [57], Vulcan [68], DynamoRIO [10], Valgrind [50], and SecondWrite [52] binary modification toolkits allow users to modify binaries at the instruction level. EEL, Etch, and Vulcan allow users to both remove original instructions and insert new code anywhere in the binary; DynamoRIO also provides the ability to modify an instruction, such as to change an operand. These tools use platform-specific instruction representations. This approach allows the user to precisely specify which instructions to use. However, it also requires the user to understand the idiosyncrasies of these instructions to ensure that they do not cause undesired side-effects, such as altering processor status flags, and limits the portability of tools written for that particular architecture.

Valgrind and SecondWrite provide similar capabilities but use a platform-independent instruction representation. Valgrind translates original instructions into a RISC-like language called UCode. All user modification is performed on the UCode representation instead of the original binary, and the modified UCode is compiled into a new binary. SecondWrite performs a similar translation into the LLVM [37] compiler's intermediate representation. These approaches remove the need for instruction set-specific knowledge by users and provide portability.

Instruction-level modification provides fine-grained control over the binary by allowing users to modify individual instructions. However, these toolkits do not provide any validity guarantees, instead relying on the user to ensure that any modifications are valid.

CFG Modification Interfaces

The DIABLO [18] binary rewriter provides both CFG and instruction modification capability through its visual LANCET [72] tool. LANCET presents the binary as a CFG. Dataflow analysis results, such as liveness information, are presented as an annotation of the graph, and colors indicate the frequency of execution as determined by profiling the binary. LANCET allows the user to arbitrarily change the CFG by changing the source or target of a control flow edge, adding new basic blocks, and splitting blocks. Furthermore, users can insert, modify, or remove instructions inside blocks.

By combining instruction-level and CFG modification, LANCET provides a platform-independent, abstract binary modification capability. However, their approach has three weaknesses. First, since LANCET operations allow the user to arbitrarily transform the CFG, they do not necessarily preserve validity of the graph; instead, they rely on the user to do so. Second, LANCET's instruction-level modifications are performed using a platform-specific representation and thus the user must have platform-specific knowledge as with instruction-level modification toolkits. Third, they do not keep the CFG and instruction representations consistent; any modification of instructions is not reflected in the CFG and vice versa. This can be dangerous if the user modifies both representations, or if one representation is used for analysis after the other is modified. For example, if a user modifies a branch instruction to target an instruction in the middle of a block, that block will not be split in the CFG.

We build upon the ideas provided by LANCET and address these weaknesses in this work. For example, we restrict the code users can insert to not include control flow instructions, such as branches, calls, and returns, thus ensuring that instruction-level modification does not alter the CFG; instead, such operations are performed on the CFG directly.

Call Graph Modification Interfaces

The Dyninst 7.0 [11] and PIN [41] binary modification toolkits can modify a binary at the function level by modifying the targets of function calls or replacing functions entirely. Conceptually, these are modifications of the program's call graph. The call graph is a multigraph that represents flow of control between different functions; nodes represent functions, and edges represent calls between functions. Dyninst 7.0 and PIN support function call replacement, which redirects an edge in the call graph to another node; function replacement, which redirects all edges to a particular node with a replacement node; and function wrapping, which refines function replacement to allow specific calls to access the original function. These replacements are specified by the user and may be compiled from a high-level language, such as C or C++.

Function modification provides a coarser granularity of modification than either instruction or CFG modification. While it is possible to modify a particular location, such as a particular branch or memory access, by replacing the entire function, doing so requires the user to construct the replacement function manually. Instead, function modification lends itself well to situations where functionality should be modified at a large granularity, such as replacing or wrapping `malloc` or similar functions. Unlike instruction or CFG-level modification approaches, operating at the function level preserves validity of the underlying CFG.

Compile-Time Modification

Compiler toolkits support modification of a program during compilation. The SUIF compiler toolkit [75] provides the Machine SUIF library [29], which allows modification of both the CFG and individual instructions. Machine SUIF provides block cloning, block creation, and edge redirection transformations. These are intended to support code layout optimizations such as loop unrolling. The library also automatically updates control flow instructions to correspond with changes to the CFG.

Unlike LANCET, Machine SUIF limits the transformations that can be applied to the CFG and thus ensures that validity is preserved; we leverage this concept and expand on it in this work. We also leverage their concept of performing control flow modification solely on the CFG instead of allowing direct user modification of control flow instructions to ensure that the CFG stays consistent with the underlying instruction representation. Machine SUIF provides basic transformations that can modify existing control flow paths. We provide additional transformations that create or remove paths while preserving the validity of the graph.

2.2 Function and Loop Abstractions

Several binary modification toolkits, including Atom [22], EEL [36], Dyninst 7.0 [11], Etch [57], Vulcan [68], PIN [41], PEBIL [38], and SecondWrite [52], describe the binary in terms of high-level function and loop abstractions. In contrast to the low-level abstractions of individual instructions and basic blocks, which have generally agreed upon definitions, previous research has developed several function and loop abstractions; these abstractions have increased in complexity over time to handle the code produced by modern compilers. In this section, we discuss the abstractions for functions and loops used by current binary modification toolkits and the techniques these toolkits use to map modification or instrumentation of these abstractions to the underlying code.

When evaluating a function abstraction, we consider the following examples of the complexities of code generated by a modern compiler:

- Shared code. Multiple functions may share the same blocks of instructions, such as error handling or teardown code. Examples of shared code are found in `libc` on Linux and the floating-point restore macros on AIX.

- Multiple entry functions. Languages may allow a function to be entered at multiple locations, such as the Fortran `ENTRY` statement. Such code generates functions that have multiple entry points.

- Interleaved functions. Compilers may outline infrequently executed code to improve cache performance. As a result, code from multiple functions may be interleaved.
- Internal entry points. As a result of outlining, the entry of a function may not be the instruction with the lowest address. This occurs when code is outlined to an earlier location in the binary.

ATOM, EEL, Etch, Vulcan, PIN, and PEBIL represent a function as a contiguous sequence of basic blocks with possible gaps for data. A particular block may only belong to one function, and functions may not be interleaved. This representation is typically derived from the information present in the symbol table, which specifies a function's name, starting address, and size. The function's entry is considered to be the instruction with the lowest address, and function exits correspond to return instructions. This approach has the advantage of simplicity, but it is not sufficient to handle the complexities of code generated by a modern compiler, as it does not handle shared code, multiple entry functions, interleaved functions, or internal entry points. In addition, entry and exit instrumentation are mapped to single instructions, and thus may be incorrect.

Dyninst 7.0 represents functions as a subgraph of the CFG with a single entry block. Function entries are identified with the symbol table (if present), by identifying the targets of other function calls, or with heuristics that identify common function preambles [25]. The body of each function is identified by performing a forward search of the CFG until a return instruction or tail call sequence is reached; these are identified as the exits of the function. This abstraction supports the examples described above. Functions may share basic blocks; in this case, Dyninst 7.0 uses per-function logical copies of such blocks. Multiple-entry functions are represented as a collection of single-entry functions that share code. Interleaved functions are possible, as the function body may be non-contiguous. Finally, the function entry is associated with a block rather than with the instruction with the lowest address.

The Dyninst 7.0 function abstraction has two weaknesses. First, Dyninst 7.0 uses the same simple mapping of entry and exit points to instructions that other toolkits use, and thus may incorrectly instrument optimized code. The second weakness is more subtle. Dyninst 7.0 handles shared code by making multiple logical copies of overlapping blocks. However, the toolkit will always make multiple copies of instrumented shared blocks, even if the instrumentation is identical for each function that contains the block. This can lead to an increase in the size of the instrumented binary if overlapping code is common.

SecondWrite extends the abstraction used by ATOM and other toolkits to allow logically distinct functions to overlap. This approach successfully represents both shared code and multiple-entry functions; the latter are represented as a set of single-entry functions. However, SecondWrite's abstraction does not allow interleaved functions or internal entry points.

An alternative abstraction is used by the PLTO whole-program optimization engine [63]. The PLTO function abstraction supports multiple entry points. Thus, if multiple functions share code, they are merged into a single function with multiple entry points. This approach avoids creating multiple logical copies of blocks as required by the Dyninst 7.0 function abstraction. However, if the code sharing is the result of an optimization rather than a language feature, this abstraction can be confusing.

EEL and Dyninst 7.0 both provide loop abstractions. These projects use natural loops [1]. EEL represents a loop as a single header block, a set of exit edges, and a set of back-edges. Dyninst 7.0 explicitly specifies the loop body as well. Loop entries are instrumented by instrumenting the edges into the header block from outside the loop, and exits are similarly instrumented at the exit edges of the loop. Finally, per-iteration instrumentation is performed by instrumenting loop back-edges. This approach does not map loop instrumentation to specific instructions, and thus does not suffer the same mapping problem as functions.

2.3 Binary Code Replacement

In this section, we describe the code replacement techniques used by binary modification and instrumentation toolkits. As with the structural representations discussed in the previous section, there are a small number of techniques shared by binary modification toolkits: patch-based, JIT, or in-place code replacement. We describe each of these techniques below. We characterize these approaches by the techniques they use, and whether they impose proportional cost; that is, whether overhead is incurred when executing original code that was not modified by the user.

Patch-based Code Replacement

Patch-based code replacement operates by appending modified or instrumented code to the binary and overwriting original code with *interception branches* to the corresponding locations in the appended code. Thus, the program executes a mixture of original and added code.

EEL [36], Dyninst 7.0 [11], Vulcan [68], PIN [41], PEBIL [38], SecondWrite [52], and self-propelled instrumentation [44] use patch-based code replacement. Vulcan uses patch-based replacement for dynamic instrumentation and in-place replacement for binary rewriting, and PIN supports both patch-based and JIT replacement for dynamic instrumentation. We characterize these toolkits as follows: first, whether they support dynamic instrumentation, binary rewriting, or both; second, the granularity at which they intercept control flow (per function, basic block, or instruction); third, whether they provide a mechanism other than interception branches to capture control flow.

EEL [36] supports binary rewriting of SPARC/Solaris programs. EEL copies the entire code region of the original binary and overwrites every instruction in the original with a branch to the corresponding location in the copy; this allows them to intercept execution on a per-instruction basis. This technique is effective on SPARC and other fixed-length architectures where a single instruction can always be overwritten with a branch. The LEEL [77] toolkit applied EEL's technique to IA-32, but relied on traps instead of branches and thus suffered high overhead.

EEL uses an interesting technique to handle jump tables, a form of indirect control flow often used to implement multiway branches such as the C `switch` statement. Jump tables map the value of an index variable to an entry in a table of destination addresses and then branch to this calculated address. EEL uses compiler-specific heuristics to identify this table and updates these addresses to point to the replacement code instead of the original code. If this update fails, EEL instead uses the original destination address and intercepts execution with the branches patched over the original code. Since the time that EEL was developed (1996), code generation strategies have become much more sophisticated, so the variations in jump table layouts have increased dramatically. As a result, heuristic-based strategies similar to EEL's are less effective today.

Dyninst 7.0 [11] supports both dynamic instrumentation and binary rewriting with the same tool infrastructure. They only copy modified or instrumented code and do so on a basic block granularity. If a block is not large enough to contain an interception branch, Dyninst 7.0 instead attempts to relocate the function that contains the block, padding out each block in the relocated function with enough space to contain an interception branch. Function relocation fails if the function includes an indirect branch with an unknown destination; in these cases Dyninst 7.0 marks the function uninstrumentable.

As a result of function relocation Dyninst 7.0 may execute two levels of interception branches; the first to a relocated function, then within that relocated function to a relocated block. Dyninst 7.0's approach is platform independent and has been implemented on SPARC, PowerPC, IA-32, x86-64, and IA-64 architectures; however, its two-level branching scheme can impose higher overhead than approaches such as EEL that execute only one level of branching.

Two other toolkits, the PEBIL binary rewriter [38] and the Vulcan dynamic instrumenter [68], use techniques similar to Dyninst 7.0. PEBIL [38] initially relocates at the instruction level; as with Dyninst 7.0, PEBIL may relocate a function if instructions cannot be patched with branches and will fail if an unparsed indirect branch is present. Vulcan's [68] dynamic instrumentation mode uses both block and function replacement, similarly to Dyninst 7.0; Vulcan relies on information provided by the compiler to determine the targets of indirect branches and thus can always relocate a function. Vulcan also supports binary rewriting but uses in-place replacement to do so; we discuss this mode in Section 2.3.

SecondWrite [52] supports binary rewriting. Like EEL, SecondWrite copies the entire binary, but only inserts interception branches at function entries. SecondWrite handles indirect control flow by inserting address translation code immediately before an indirect branch or call. This code converts the original destination of the indirect control flow instruction to its corresponding address in the replacement code. This scheme obviates the need for block-level interception branches, but imposes higher overhead since executing address translation code is typically more expensive than executing an interception branch.

These toolkits all operate as third-party programs that modify the target binary, either on disk or at run-time via the debugger interface. Patch-based replacement has also been applied from *within* the executing program with a technique called self-propelled instrumentation [44]. This technique replaces call sites with interception branches to code snippets that contain both user-specified instrumentation code and toolkit-provided propagation code. This propagation code identifies the callee and instruments it. As a result, instrumentation is propagated along execution paths. This approach is amenable to situations where a third-party instrumentation may not be feasible, such as distributed systems, and often results in lower overhead by avoiding a context switch between the executing program and the toolkit.

Finally, the PIN [41] toolkit provides partial support for patch-based code replacement. PIN is primarily a JIT-style toolkit as described in the next section. However, PIN supports a patch-based "probe mode". This mode supports instrumenting only function entries and exits, and operates by copying whole functions. These copied functions cannot contain indirect control flow and must be represented in the symbol table.

Patch-based replacement will impose proportional cost in space and time if only modified or instrumented code is copied; thus, Dyninst 7.0, PEBIL, Vulcan, and PIN impose proportional cost while EEL and SecondWrite do not. Dyninst 7.0 and Vulcan support both dynamic instrumentation and binary rewriting with the same tool infrastructure.

JIT Code Replacement

JIT code replacement begins by identifying the current execution point in the process and copying a sequence of instructions beginning at that point into a *code cache*. This trace is then instrumented or modified as desired by the user. When the trace finishes executing, control is transferred back to the toolkit, which identifies, copies, modifies, instruments, and executes the next trace. As a result, no original code is ever executed; instead, execution occurs entirely in the code cache and the toolkit itself.

Shade [14], DELI [19], DIOTA [42], and PIN [41] are JIT-based dynamic instrumenters; DynamoRIO [10] and Valgrind [50] provide JIT-based binary modification in addition to instrumentation. Unlike patch-based code replacement, there is a great deal of similarity between the various JIT-based tools. Therefore, we only mention unique aspects of these tools.

Shade was the first JIT instrumentation toolkit, and was remarkable for performing binary translation as well as instrumentation. Shade could translate between SPARC (v8 and v9) and MIPS by emitting code for a different architecture than the original code. However, Shade's traces were single instructions, and thus its overhead was high since the JIT engine was called frequently.

DIOTA developed a mechanism for handling self-modifying code by write-protecting code pages and intercepting write operations; this approach was also used by DynamoRIO and PIN. This approach works well if code and data are separated, but leads to high overhead if code and data are mixed on the same pages.

PIN focuses instead on optimization of the generated code and pioneered several address translation optimizations, such as code cloning to enable constant propagation, that have been reused by other JIT-based toolkits such as DynamoRIO.

JIT code replacement does not require a static parse of the binary, and thus works transparently both on programs with substantial indirect control flow and programs that generate code at run-time. Also, since toolkits that use this approach do not overwrite original code, they do not create problems for self-checksumming and similar tamper resistance techniques. However, this approach has three weaknesses. First, since all executed code is copied and relocated, these toolkits impose cost even when executing unmodified code; thus, they do not impose proportional cost. Also, the cost involved with executing the JIT engine can dominate if the program is short-lived, although it may be amortized for long-running programs. Second, these tools handle indirect control flow using an address-translation approach similar to SecondWrite: the program calculates the original destination, and it is translated to a replacement destination immediately before the indirect branch or call. This approach imposes higher overhead than using an interception branch. Third, this approach is not amenable to binary rewriting since it relies on a dynamic parse of the program.

In-Place Code Replacement

In-place replacement operates by moving original code to make space to insert new code and updating any address calculations to correspond to the new structure of the binary. This technique imposes lower overhead than both patch-based and JIT code replacement, since there is no need for interception branches, address translation, or a JIT engine. ATOM [22], Etch [57], Vulcan [68], and DIABLO [18] use this technique to provide binary rewriting.

However, this technique relies on the presence of linker relocations to allow the toolkit to update address calculations. Conceptually, linker relocations provide sufficient information about the binary to make all code position-independent; however, this information is normally only present in object files or statically linked code and is discarded after it is used. Thus, this approach can only be applied to specially prepared binaries. Furthermore, in-place code replacement is not amenable to dynamic instrumentation, since the information provided by linker relocations is not sufficient to update executing process state.

2.4 Behavior Preservation

The code replacement strategies discussed in the previous section alter the binary by copying original code, overwriting original code, and adding new code. This section discusses approaches for compensating for the effects of these operations on the binary to ensure the original binary's behavior is preserved. We also describe a related technique that compensates for the effects of executing instrumentation code on shared library state.

Binary modification toolkits modify the structure of the binary by moving original code, adding new code, and overwriting original code. These operations can affect the execution of *sensitive* instructions. Conceptually, an instruction is sensitive if one or more of its inputs will be changed by a modification; we formalize this concept in Chapter 7. For example, a load instruction that treats code as data would be sensitive to overwriting that code with new values. Similarly, an instruction that uses the program counter (PC) would be sensitive to movement because the value of the PC would change. We discuss approaches for handling overwritten or moved code below; to our knowledge, no binary modification toolkits attempt to compensate for instructions sensitive to added code.

The simplest method for preventing behavior changes is to use a modification technique that does not modify program structure. This approach is used by JIT-based toolkits [10, 14, 19, 41, 42, 50] that do not overwrite any original code. While this approach suffices for programs sensitive to overwritten code, such as a self-checksumming program, it does not compensate for moved code, which will perceive a different program counter value, or added code, since adding code changes the shape of the address space.

Wurster *et al.* [76] compensate for overwritten code by redirecting instruction fetches and data accesses to different regions of memory by manipulating the translation lookaside buffer (TLB). This TLB splitting approach makes a copy of the original code and redirects data accesses to that copy while executing the modified code; as a result program behavior is preserved with no overhead. TLB splitting has been implemented by modifying the kernel [76] or a virtual machine monitor (VMM) underlying the kernel [59]. Unfortunately, it can be defeated by combining self-modifying code with self-checksumming code [23], since writes occur only on the data copy. Thus, when the code attempts to modify itself, it will update the data copy but not the code copy and the modifications will not be reflected in future instruction fetches.

The most common technique for compensating for the effects of moved code is to emulate the original value of the program counter. For example, a call instruction would be split into a sequence that saved the original return address and then branched to the destination of a call. As a result, all addresses are calculated relative to the original program. This approach also requires address translation for all control flow pointers to code that may have been moved. This approach is used by JIT toolkits [10, 14, 19, 41, 42, 50]; these toolkits insert address translation code at all indirect branches, indirect calls, and return instructions. Our experiments show the combination of emulation and translation imposes an average execution overhead of 90%; these results are shown in Chapter 7.

However, not all changes in instruction behavior will affect overall program behavior. For example, moving a call instruction is often safe if the stored return address is only used for control flow. This intuition motivates the ad-hoc compensation technique developed by Dyninst 7.0 [11] and used by PEBIL [38]. This approach attempts to emulate only instruction sequences that copy the program counter into a general purpose register; instructions that use the PC to save a return address are left unmodified. However, this approach uses manually specified patterns to recognize sequences that require emulation, may result in false negatives, and cannot guarantee behavior preservation.

The DIABLO link-time optimization system [18] addresses a separate problem, that of preventing instrumentation code's execution from modifying data structures in shared libraries [17]. They note that instrumentation that uses library functionality, such as to allocate memory or open a trace file, may perturb applications that depend on particular patterns of library behavior, such as precise values of file descriptors or pointers to allocated memory. DIABLO provides specialized support libraries that instrumentation can use to safely access such functionality. The PLTO optimization system [2] provides similar libraries for memory management. This support library approach is complementary to our sensitivity resistant code relocation technique.

2.5 Summary

There has been significant research into binary modification and instrumentation. In this chapter we discussed four facets of this work: the interfaces used to modify binaries, the function and loop abstractions that provide a higher-level structural representation of binaries, the techniques used to replace original binary code with modified code, and the techniques used to compensate for the side-effects of code replacement on the binary.

Binary modification has been performed on instructions, the CFG, and functions. Of these, the CFG provides both fine-grained control and a platform-independent interface. However, previous CFG modification approaches fail to enforce safety constraints; it is possible to create a CFG that cannot be instantiated into binary code or an invalid CFG that describes a structurally invalid binary. We address this problem with structured binary editing, a CFG-modification technique that preserves validity, in Chapter 3.

Several function abstractions exist that trade off simplicity for effectiveness on program binaries produced by modern compilers. Current binary modification toolkits make the assumption that function entries and exits correspond to single instructions. As we showed in the introduction, this assumption is not always true; these toolkits may incorrectly instrument optimized or obfuscated code. EEL and Dyninst 7.0 provide a loop abstraction and instrument these abstractions in terms of edges, rather than instructions, thus avoiding the problems of mapping entry and exit to particular instructions. We present our extension of structured binary editing to functions and loops in Chapter 4.

Patch-based code replacement provides both dynamic instrumentation and binary rewriting, imposes lower overhead than JIT replacement, and can operate without the linker relocations required by in-place replacement. However, current patch-based methods do not provide timely replacement of actively executing code and may execute unnecessary interception branches, adding to the execution cost. We address these problems with anytime code replacement, which we present in Chapter 6.

Current behavior-preservation approaches handle moved or overwritten code; no solution for hiding the presence of added code exists. Moved code has been handled by either emulating all moved instructions, which may impose unnecessary overhead, or recognizing which instructions to emulate with pattern-matching heuristics, which may be unsafe. Overwritten code can be addressed by providing a specialized execution environment that hides overwritten code by altering the TLB; however, this approach may be ineffective if a program generates code at runtime. We present our sensitivity-resistant code relocation technique, which handles moved, overwritten, and added code, in Chapter 7.

3 Structured Binary Editing

Binary modification allows the user to alter the structure and control flow, and thus the behavior, of a binary program. In this chapter, we describe structured binary editing, a binary modification approach based on transformation of the CFG that provides platform-independent binary modification while ensuring that these modifications do not create a *structurally invalid* program binary that has illegal control flow or executes invalid instructions. The CFG is a familiar representation of a binary's structure and control flow; we demonstrate that the CFG can also be used to modify these characteristics. The core of our approach is an algebra of CFG transformations that preserve the validity of the CFG and thus ensure the modified CFG can be instantiated into a new binary that has the corresponding changes to its control flow. The operations in this algebra are closed under CFG validity, and thus these transformations can be composed to provide powerful binary modification capabilities without requiring instruction-level understanding of the binary by users.

We begin by defining our CFG representation, which is derived from the CFG representation used by the ParseAPI component of Dyninst 7.0 [54]. A CFG is a directed graph whose nodes represent *basic blocks* and edges represent control flow between blocks. A basic block is a contiguous sequence of instructions that execute as a single unit; blocks are commonly used to reduce the size of the graph. We also define abstractions for *functions* and *loops*. In this chapter we only present transformations of the CFG; we discuss both modification and instrumentation of functions and loops in the next chapter.

We then define program structural validity and CFG validity, using the

CFG definition presented in Section 3.1. Conceptually, a program is structurally valid if it will execute valid instructions, and a CFG is valid if it represents a structurally valid program. CFGs that are derived from structurally valid programs by correct parsing techniques are implicitly valid; however, this characteristic may not hold if the graph is transformed without constraint. We define CFG validity as a constraint over each element in our CFG definition. Our approach enforces structural validity of the resulting binary; other, stronger forms of validity, such as enforcing alignment of the stack, are beyond the scope of this work.

Next, we define our algebra of CFG transformations, building on the CFG transformations provided by LANCET [72]. We demonstrate that the operations of this algebra preserve CFG validity and thus an arbitrary composition of transformations will preserve validity as well. We define three classes of transformations: block transformations that alter basic blocks; edge transformations that redirect or otherwise modify control flow edges between blocks; and code insertion transformations that either insert a predefined code sequence, such as a call, return, or conditional branch, or a user-provided sequence of new code.

We then discuss the problem of handling indirect control flow. Our initial treatment considers only direct control flow where the validity of a transformation can be immediately determined. Indirect control flow, however, depends on the data flow of the program; this, in turn, can be altered by modifying control flow. We present a slice-based dataflow analysis that identifies which indirect control transfers (ICTs) may be affected by transforming the CFG, and attempt to determine the new possible destinations of these transfers using the same techniques we use to parse the initial CFG. If this succeeds, we can determine whether the transformation will preserve validity and disallow transformations that do not. If the analysis fails, we instead insert runtime verification code that dynamically ensures validity.

Next, we discuss the implementation of our techniques in the Dyninst 7.0 binary analysis and instrumentation toolkit. When compared to other binary modification toolkits Dyninst 7.0 has two clear advantages. First, it uses the CFG as its primary program representation, and thus gave us the foundational abstractions on which to build. Second, it provides the analysis capabilities (e.g., slicing) required for our dynamic validity analysis as described in Section 3.4. However, implementing binary modification in Dyninst 7.0 posed challenges as well. While Dyninst 7.0 provides a CFG representation of the binary, this CFG is essentially static. We extended Dyninst 7.0 in three ways: first, by making its CFG user-modifiable; second, by allowing insertion of new code into the CFG; and third, by extending its code generation to instantiate a new binary from the modified CFG.

We conclude the chapter with a presentation of two case studies that demonstrate the efficacy of our techniques. We first present a tool that applies security fixes to a running Apache web server, and compare this tool to similar approaches using instruction-level binary modification. Second, we describe ongoing work that that alters the precision of floating-point instructions. This effort is being performed by Michael Lam at the University of Maryland [35], and relies on our structured binary editing to enable alteration of floating-point instructions.

3.1 Control Flow Graph

The CFG represents the structure of a binary and describes possible execution paths. Our CFG is based on five abstractions: the *interprocedural control flow graph* that consists of *basic blocks* and *edges*; *functions*; and *loops*. This graph provides a platform-independent representation of the structure of the underlying binary. We define our abstractions for functions and loops in Chapter 4. In addition, we define *code snippets* to represent code that a user adds to the binary.

A CFG is a directed graph $G = (V, E, V_e, V_x, T)$, defined as follows; we show an example CFG in Figure 3.1:

- The set $V = B \cup \{v_\perp\}$ of vertices corresponding to basic blocks B and a *sink* $v_\perp$,

- The set $E \subseteq V \times V$ corresponds to control flow edges between blocks,

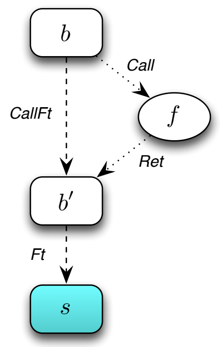


Figure 3.1: An example CFG that demonstrates our notation. Blocks (b, b', s) are represented by boxes and edges as arrows that are labeled with their types. We shade snippet blocks, such as s, in blue. Dashed arrows represent *intra*procedural edges, while dotted arrows represent *inter*procedural edges. We summarize functions as ellipses (f).

- The sets $V_e \subseteq V$ of entry vertices and $V_x \subseteq V$ of exit vertices, and
- The function $T : E \rightarrow \mathcal{T}$ that associates edges with *types*.

We define basic blocks in the conventional way as a consecutive sequence of instructions $b_i = \langle i_m, \ldots, i_n \rangle$ with a single entry instruction i_m and single exit instruction i_n; an instruction may belong to only one block. Each instruction in a block is pre-dominated by its predecessor and post-dominated by its successor. The in- and out-edges of b_i are denoted $In(b_i)$ and $Out(b_i)$. Unknown control flow is represented by an edge to a unique *sink* $v_\perp$ that contains no instructions and has no out-edges.

Edges are associated with a *type* that is an element in the following set:

$$\mathcal{T} = (Dir, Ft, Cond, CondFt, Ind, Call, CallFt, Ret)$$

that represents *direct, fallthrough, conditional taken, conditional fallthrough, indirect, call, call fallthrough*, and *return* edges, respectively. Call fallthrough edges link blocks ending with calls to their intraprocedural successors, and may be omitted if the callee does not return. Call and return edges are *interprocedural*; all other edges are *intraprocedural*.

Our CFG model has been successfully applied to IA-32/x86-64 and PowerPC. It is derived from the Dyninst CFG as provided by the ParseAPI component [54], which has also been applied to SPARC and IA-64. We expect that the CFG will also be able to represent ARM binaries. ARM is an interesting architecture in this respect because of its pervasive use of predicated control flow instructions, including calls and returns. However, PowerPC has a similar capability (which we discuss in detail in Section 4.1) and so we expect supporting ARM will be a straightforward process.

Deriving a CFG from a binary is a difficult problem in its own right. We assume the presence of a parsing algorithm that can derive a CFG from the binary. The design of these algorithms is a challenge due to the presence of both indirect control flow that cannot be statically analyzed and data intermixed with code. We use the ParseAPI [54] recursive-traversal parser [13, 64] that follows statically determinable control flow to discover as much code as possible, and makes use of backwards slicing and heuristic techniques to identify the targets of indirect jumps (e.g., jump tables [12]) and functions that are only reached via indirect calls [25]. To our knowledge, this algorithm has never been published.

The CFG resulting from this parsing may be incomplete due to unknown indirect control flow, exceptions, and similar constructs. For conventional binaries, this incompleteness can be addressed by making conservative assumptions about such control flow, such as that any function may be the target of an indirect call, and that indirect branches are intraprocedural. In such cases, additional edges can be added to the CFG to represent these assumptions, and we may mark functions as unmodifiable if they contain unknown control flow. If the user is modifying a running process, we may augment our static parse with dynamic analysis techniques, such as those described by Roundy and Miller [60]. These techniques will always present a locally complete CFG that can be safely modified.

Users insert additional code into the CFG by providing a *code snippet*. A code snippet is a single-entry, single-exit sequence of code that may contain internal branching. We use this definition, rather than the finer-grained definition of a snippet as a single block, to provide maximum compatibility with existing tools that generate such regions of code, such as LLVM [37] or Dyninst 7.0's AST [11] and DynC [53] code generation languages. We do not allow control flow edges from within the snippet to external targets, such as a call to an existing function, to ensure that inserting new code does not modify the surrounding blocks and edges; such modification must instead be performed with an explicit additional CFG transformation. Users may construct more complex code sequences, such as code that ends in a conditional branch or makes a call, by combining multiple snippets with the appropriate CFG transformations. Once a snippet has been inserted into the CFG, it is represented as a subgraph of the CFG and may be further transformed using the same transformations that can be used on the original CFG.

We define a code snippet s_j as a CFG $(V_j, E_j, v_j, x_j, T_j)$ with a single entry block v_j and exit block x_j; the exit block must have a single fallthrough edge to the sink $v_\perp$. Code snippets may contain internal branching, but may not have explicit branches outside of the snippet. For simplicity, we presume that each snippet is specified as a CFG. They could also be provided as a buffer of assembly that is parsed at insertion time to create a CFG, or in some representation that can be compiled to assembly.

3.2 Validity

Structured binary editing allows the user to modify a binary program while ensuring that its structural validity is preserved. Instead of directly modifying the program, we derive a CFG from the binary, transform the CFG, and use the transformed CFG to instantiate a similarly modified program. By ensuring that all transformations preserve the validity of the CFG, we ensure the resulting program is structurally valid. In this section, we define structural validity of binary programs and validity of CFGs. Intuitively, a binary program is *structurally valid* if it will only execute valid instructions; structural validity does not guarantee correct execution or output. Similarly, a CFG is *valid* if it represents a structurally valid binary program and can thus be used to instantiate such a program.

b_1
Dir
b_2
b_1
b_2

(a) Original CFG (b) Transformed CFG

Figure 3.2: An example of an invalid CFG transformation. Figure (a) shows the original CFG, with $V = \{b_1, b_2\}, V_e = \{b_1\}, V_x = \{b_2\}, E = \{(b_1, b_2)\}$, and T mapping this edge to type Dir. Figure (b) shows the CFG with the edge (b_1, b_2) removed. The second CFG is invalid, as b_1 has no out-edges and is not an exit block, and cannot be instantiated as a structurally valid binary program. Thus, edge removal does not satisfy our constraints and is thus an invalid transformation.

Structural Validity

For our definition of structural validity, we represent a binary program as a tuple $P = (C, D)$ where $C = \langle i_0, \dots, i_m \rangle$ is a sequence of instructions representing the binary code and D represents data. We say P is *structurally valid* if for all inputs, P will only execute instructions from C, and does not treat values from D as instructions or execute unallocated memory. We purposefully use a permissive model of validity that allows constructs such as code in data (as C and D may overlap) and overlapping instruction sequences, which is a common code obfuscation concept.

We assume the input program P is valid. This may not be the case, either if P contains code that is modified at runtime by the remainder of the program or if it contains intentionally invalid instructions. In such cases, our approach will preserve the invalidity of the program.

CFG Validity

A CFG is implicitly valid when it is first derived from a structurally valid binary, since its blocks describe valid instruction sequences and edges represent the control flow paths the binary will take. However, unconstrained transformation of the CFG may result in a graph that no longer represents a structurally valid binary program. Consider the simple example shown in Figure 3.2, where a transformation has removed all of the out-edges of a block b_1. As a result, b_1 has no identified successors; since b_1 is not an exit node, the graph does not represent a structurally valid binary program and thus cannot be used to instantiate one. We use our definition of CFG validity to constrain which graph transformations can be applied to a CFG.

We define CFG validity with *constraints* over each element in the CFG definition. Let $P = (C, D)$ be a structurally valid program as defined above. Then a control flow graph $CFG_P = (V, E, V_e, V_x, T)$ for P is valid if the following constraints hold:

- *Vertex constraint*: The vertex set $V = B \cup \{v_\perp\}$ is valid if every block $b \in B$ is valid by the definition of basic blocks and the sink $v_\perp$ is unique.

- *Edge constraint*: The edge set E is valid if no edge has the sink as a source and all non-exit blocks $b \in (V \setminus V_x)$ have at least one out-edge. That is, E is valid iff $\nexists e \in E$ s.t. $e = (v_\perp, b_i)$ for some block b_i and $\forall b_j \in (V \setminus V_x), \exists e \in E$ s.t. $e = (b_j, b_k)$ for some block b_k.

- *Entry constraint*: The entry set V_e is valid iff $V_e \subseteq V$, $|V_e| \geq 1$, and $v_\perp \notin V_e$.

- *Exit constraint*: The exit set V_x is valid iff $V_x \subseteq V$ and $v_\perp \notin V_x$. Unlike the entry set, the exit set may be empty if the program never terminates.

- *Type constraint*: The type function T is valid if all edges have a type, and the out-edges of each block b are labeled in a way that corresponds to the appropriate architecture. This constraint is architecture-specific because each architecture has different control flow instructions with different characteristics; for example, PowerPC and ARM support conditional indirect branches while IA-32 only supports unconditional indirect branches. We represent valid edge types, or *signatures*, in disjunctive normal form, and we define signatures for IA-32/x86-64, PowerPC, and ARM in Table 3.1. We have implemented support for IA-32/x86-64 and PowerPC; an ARM implementation is in the planning stage.

IA-32/x86-64	PowerPC/ARM
$(\mathit{Dir}) \vee$	$(\mathit{Dir}) \vee$
$(\mathit{Ft}) \vee$	$(\mathit{Ft}) \vee$
$(\mathit{Cond} \wedge \mathit{CondFt}) \vee$	$(\mathit{Cond} \wedge \mathit{CondFt}) \vee$
$(\mathit{Ind}^{+}) \vee$	$(\mathit{Ind}^{+}) \vee$
$(\mathit{Call}) \vee$	$(\mathit{Call}) \vee$
$(\mathit{Call} \wedge \mathrm{CallFt}) \vee$	$(\mathit{Call} \wedge \mathit{CallFt}) \vee$
(Ret^{+})	$(\mathit{Ret}^{+}) \vee$
	$(\mathit{Ind}^{+} \wedge \mathit{CondFt}) \vee$
	$(\mathit{Call} \wedge \mathit{CondFt}) \vee$
	$(\mathit{Call} \wedge \mathit{CallFt} \wedge \mathit{CondFt}) \vee$
	$(\mathit{Ret}^{+} \wedge \mathit{CondFt})$

Table 3.1: Valid signatures for IA-32/x86-64, PowerPC, and ARM. In some cases multiple out-edges may have the same type; we label these with $^{+}$, borrowing notation from regular expressions. For example, one or more indirect out-edges is valid. A call will have a call fallthrough edge if the callee returns; for non-returning calls, such as `exit`, we omit the call fallthrough edge. PowerPC and ARM support conditional indirect branch, call, and return instructions; this complicates the signature for this architecture.

Let $\mathit{Out}(b) = \{e_i, \ldots, e_j\}$ be the out-edges of a block b, and T_b a sequence of types $\langle T(e_i), \ldots, T(e_j)\rangle$; we represent T_b as a sequence since multiple edges may have the same type, such as for an indirect branch. Each sequence T_b is valid if it satisfies the appropriate signature, and T is valid if $\forall b, T_b$ is valid.

As examples, let b be a block with two out-edges e_1, e_2. A type function T_1 that maps $e_1 \rightarrow \mathit{Cond}$ and $e_2 \rightarrow \mathit{CondFt}$ is valid on IA-32, x86-64, and PowerPC. A type function T_2 that maps $e_1 \rightarrow \mathit{Call}$ and $e_2 \rightarrow \mathit{CondFt}$ is valid on PowerPC or ARM but not on IA-32 or x86-64. Finally, a type function that maps $e_1 \rightarrow \mathit{Cond}$ and $e_2 \rightarrow \mathit{Dir}$ is not valid on either architecture since there is no control flow instruction that can produce both a conditional and direct out-edge.

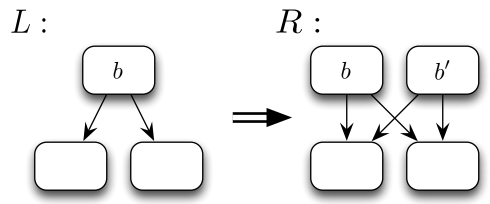


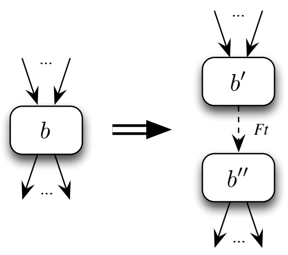
Figure 3.3: An example graph transformation. The input subgraph L is on the left, and the replacement R is shown on the right, separated by a double arrow. We use solid arrows to indicate an edge that can be either intra- or inter-procedural, and omit type labels when they are not necessary. Edges to or from blocks not included in the transformation (such as the source edge(s) of b) are omitted for clarity.

3.3 CFG Transformation Algebra

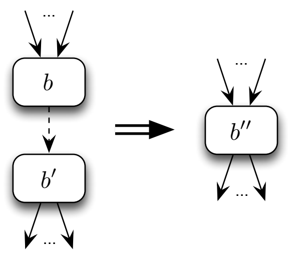
The major contribution of this chapter is an algebra of graph transformations for editing the structure and control flow of a binary program. In this section, we define this algebra. We then describe three classes of transformations: *block*, *edge*, and *code insertion*, and provide examples of each class. This discussion assumes that each transformation has no other effect on the CFG; specifically, that indirect control flow is not changed. We discuss our approach to handling indirect control flow in Section 3.4.

Our structured binary editing algebra is a tuple $BinEdit = (GT, VC)$ where GT is a set of graph transformation *rules* and VC represents the validity constraint defined in Section 3.2. Briefly, a rule $r : L \rightarrow R$ replaces an instance of the subgraph L in a target graph G with the graph R; we represent these rules graphically, as is typical in graph transformation [3, 26]. A rule r is *valid* under the constraint VC if transforming an input graph that satisfies VC results in an output graph that satisfies VC. We show an example graph transformation to demonstrate our notation in Figure 3.3. Users may compose these transformations; composition of two valid rules is also valid.

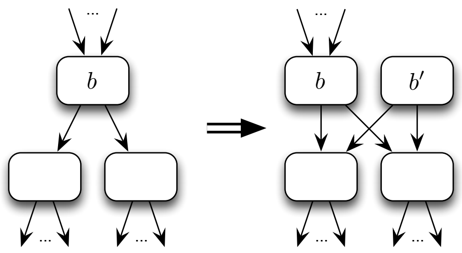
The transformations presented in this section are valid so long as each transformation has no effect on the surrounding CFG. This may not be the case if a transformation alters indirect control flow, and we address this problem in



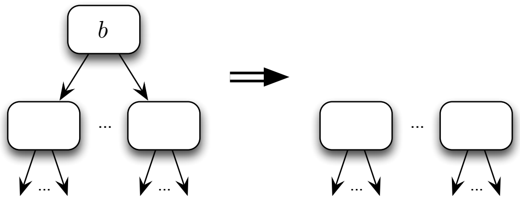
(a) Block Splitting



(b) Block Joining



(c) Block Cloning

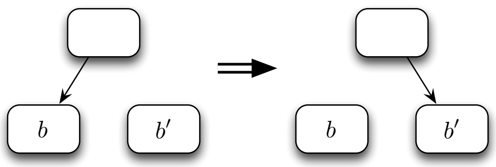


(d) Block Removal

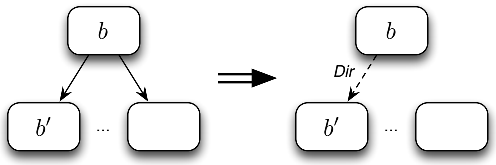
Figure 3.4: Block transformations. Block splitting is shown in figure (a); the original block $b = (i_1, \ldots, l_n)$ is split into two blocks $b' = (i_1, \ldots, i_m)$ and $b'' = (i_{m+1}, \ldots, i_n)$. Block joining is shown in figure (b), where two blocks b, b' are combined into one block $b'' = b \cup b'$; note that b' may have no other in-edges except from b. Block cloning is shown in figure (c), where the block b is cloned creating a block b'. Note that cloning copies out-edges but not in-edges. Finally, we show block removal in figure (d); note that the removed block may not have in-edges.

Section 3.4.

The first class of transformations alter the blocks (nodes) in the CFG. We define four block transformations: *block splitting*, *block joining*, *block cloning*, and *block removal*. These transformations are shown in Figure 3.4. Block splitting divides an existing block into two pieces and joins the resulting blocks with a



(a) Edge Redirection

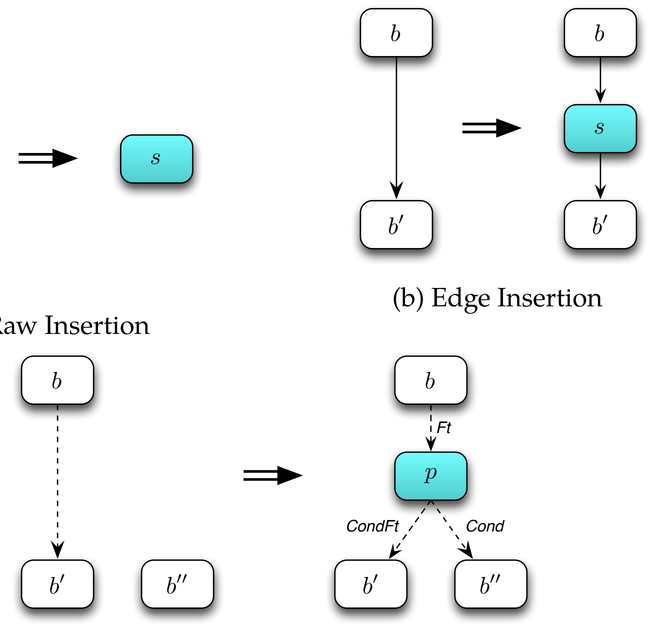


(b) Edge Collapsing

Figure 3.5: Edge transformations. Edge redirection is shown in figure (a); the target of the edge is changed from b to b' with no other changes. Edge collapsing is shown in figure (b); all out-edges of b are replaced by a single out-edge to b'. All blocks may have in-edges from blocks not included in the transformation; however, blocks with no in-edges are valid.

fallthrough edge. Blocks must be split at an instruction boundary. Block joining reverses this operation. The first block must have a single intraprocedural out-edge that targets the second block; thus, this edge must be typed as either direct or fallthrough. The second block must have a single in-edge from the first block. Block cloning creates a copy of a particular block, including all of that block's out-edges (but not its in-edges). Block removal deletes a block from the graph; the removed block must have no in-edges.

The second class of transformations alter the edges in the CFG. Since these transformations involve no new code, they can only alter or remove existing execution paths; we discuss creating new paths below. We define two edge transformations: *edge redirection* and *edge collapsing*. These transformations are shown in Figure 3.5. Edge redirection changes the target of an edge; the source is left unmodified. Edge collapsing replaces all out-edges of a block with a single edge to a selected target block; this edge is typed as *direct* and with a context of intraprocedural. One obvious transformation that we do not support is edge *deletion*, which removes an edge from the graph. Applying



(a) Raw Insertion

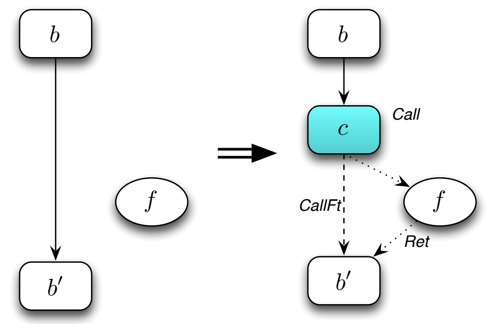
(b) Edge Insertion

(c) Predicate Insertion

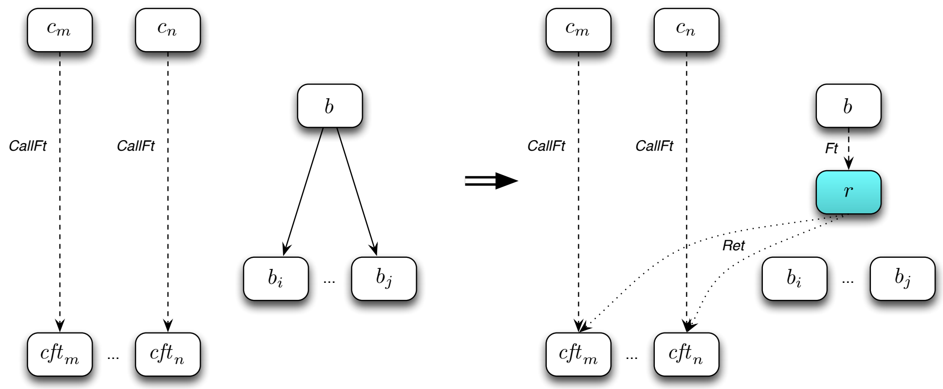
Figure 3.6: Insertion transformations, with the inserted snippet represented by blocks shaded in blue. Raw insertion is shown in figure (a); the added snippet s is not connected to the original CFG. Edge insertion is shown in figure (b); a new snippet s is inserted between b and b'. Predicate insertion is shown in figure (c); in the original graph b and b' are connected by a direct or fallthrough edge, and we introduce a conditional edge to b'' while converting the original edge to a conditional fallthrough. The block p represents the predicate that selects which edge is taken at runtime.

such a transformation may result in an invalid CFG, since it does not provide an alternative execution path (Figure 3.2). Similarly, modifying the source of an edge is invalid in our algebra since it may result in an invalid CFG.

The third class of transformations inserts new code into the CFG. We must constrain such insertion to ensure that the resulting CFG is valid; we do this by constraining the transformations that can be used to insert code. In this work we define five insertion transformations: *raw insertion*, *edge insertion*, *predicate insertion*, *call insertion*, and *return redirection*. These transformations are shown in Figures 3.6 and 3.7. Raw insertion simply inserts a provided snippet into



(a) Call Insertion



(b) Return Insertion

Figure 3.7: Insertion transformations continued, with the inserted snippet represented by blocks shaded in blue. Call insertion is shown in figure (a); we insert a new block c (shown shaded) that contains the call to f. Return replacement is shown in figure (b); we replace the out-edges of a block b in a function f with return edges to the caller(s) of f. We represent the blocks that call f as $c_m, \ldots, c_n$, and the corresponding call fallthrough blocks as $cft_m \ldots cft_n$. The shaded block r contains the code necessary for a valid return (e.g., stack teardown).

the CFG, but performs no other operations; as a result, the user must use other transformations to connect its entry and exit with the original CFG. Edge insertion inserts a snippet along an existing single control flow edge; the snippet entry is linked to the source of the edge and the exit is linked to the target. Predicate insertion creates new execution paths by converting an existing direct or fallthrough edge to a conditional pair. This transformation inserts a predicate that controls which edge is executed and a new conditional taken edge e_t, while converting the original edge to a conditional fallthrough e_{ft}. For simplicity, the predicate tests a single register or memory location: if the value is non-zero, e_t is taken; if zero, e_{ft} is taken instead. If more complex predicates are required, a user can create them by composing predicate insertion with edge insertion, and we rely on existing code-optimization techniques to generate efficient code for the desired result. Call insertion allows a user to interpose a function call along an edge; to preserve validity, the original edge must be typed as either direct or fallthrough. Finally, return redirection replaces the out-edges of a block with an immediate return to the calling function.

3.4 Indirect Control Validity

In the previous sections, we described an approach to modifying a binary program by transforming its CFG. This approach made the simplifying assumption that these transformations had no effect on the remainder of the CFG. This simplification does not hold if a program uses indirect branches or calls that use addresses determined at runtime. In this section, we describe how transforming the CFG may alter indirect control flow and describe an approach for safely determining when this may occur.

Programs rely on indirect control flow for language features such as multiway branches, function pointers, or virtual functions. At the binary level, these constructs are similar: the program first calculates an address and then uses an *indirect control transfer* (ICT) to branch or call to this destination. These address calculations may be altered by a transformation of the CFG, such as by inserting new code that changes intermediate values used by the calculation. Such an altered calculation may change the possible destinations of the corresponding

ICT. This, in turn, may result in an invalid CFG that does not properly represent these new destinations. The transformations presented in Section 3.3 do not take this case into account, instead assuming that the only control flow effects on the CFG are those explicitly represented in the transformation.

This problem is compounded by the difficulty of statically determining indirect control flow. Compiler-emitted address calculations (e.g., jump tables) frequently can be successfully analyzed and their destinations represented in the CFG [12]. However, more complex calculations, such as those used for function pointers or in optimized code, are either expensive to analyze [6] or defeat analysis entirely. We label such ICTs in our code with edges to the sink node, representing statically unknown control flow.

We identify which ICTs are affected by a particular CFG transformation and how they are affected as follows. We use static analysis to identify which ICTs are affected; clearly, if an ICT is not affected by a transformation then its destinations will not be changed. If the ICT is affected, we attempt to statically determine its new destination set. If each such destination is valid, we update the CFG to match; if a destination is invalid we inform the user and invalidate the transformation. Finally, if we fail to resolve the possible destinations via static analysis, there is a spectrum of responses: we could invalidate the transformation, allow the transformation but insert runtime monitoring code, or allow the transformation and warn the user of the CFG possibly becoming invalid. We allow the user to select which response they prefer, which allows a user to be conservative if they want while allowing transformations that the user may know is valid even if our analysis fails.

We define a destination to be valid if it targets the entry of a known basic block (for an indirect jump) or known function (for an indirect call). This is purposefully a restricted definition following our policy to minimize the possible secondary effects of a CFG transformation. More relaxed policies are possible; for example, we could define a destination to be valid if it targeted a known instruction, and implicitly split a block at such a targeted instruction.

Our analysis is performed over the transformed CFG, which may include inserted code snippets as well as original code. This analysis can be simplified if the user provides semantic *summaries* of inserted code snippets; however, such summaries are not required. When we encounter a snippet during our analysis, we apply a summary if it is available and continue instead of having to analyze the entire snippet. This simplification biases towards cases where inserted code has no cumulative effect on the address calculation (e.g., instrumentation) but proving this with analysis of the inserted code would be expensive. Specifically, if a snippet has no cumulative effect, we can omit its effects entirely and quickly determine that the ICT was not affected.

Identifying Affected Indirect Jumps

Intuitively, an ICT is affected if a CFG transformation interferes with the address calculation preceding the ICT. Let i_j be such an ICT in a function f in the original CFG, t a transformation, and f' be the resulting transformed function with corresponding ICT i'_j. We define the address calculation corresponding to i_j as the *backwards slice* ac_j, bounded at function entry [33]; we discuss our selection of this bound below. Clearly, if t has no effect on ac_j then the ICT i_j will also be unaffected; similarly, if t alters ac_j then i_j is affected.

This is essentially a dataflow analysis problem, and thus it is insufficient to see if ac_j is directly affected by t. Instead, we calculate the new backwards slice ac'_j from i'_j in f'. If $ac_j \equiv ac'_j$, then no modification of the address calculation has occurred and thus $i_j \equiv i'_j$; otherwise, we conclude i_j is affected and perform further analysis to determine its new destinations.

We bound the backwards slice at unanalyzable memory accesses or the entry of the function. Our current implementation can resolve stack accesses or accesses to fixed memory locations; all others, such as a pointer-based heap reference, terminate the slice. If we reach an unanalyzable memory access, we make the conservative assumption that the ICT was affected by the transformation. If we reach the entry of the function we can safely end the slice since an intraprocedural transformation will not alter the function's inputs.

Identifying New Destinations

Once we have identified the ICT i_j that is affected by the transformation t, we must determine its new possible destination addresses; let this be a set of addresses $D = \{a_m, \ldots, a_n\}$. We say i_j is *valid* if $\forall a_k \in D$, a_k is the entry of a block b_k. We analyze D as follows. First, we use symbolic evaluation to convert the slice ac_j into a symbolic representation $eval_j$ [62]. We then attempt to evaluate $eval_j$ to determine its possible outputs. If this evaluation succeeds, we examine each possible destination to determine if it is valid; if all destinations are valid, we update the out-edges of the ICT in the CFG and inform the user of each changed edge. If a destination is invalid, we invalidate the transformation. Finally, if the evaluation fails, we perform one of three actions depending on the user's preference for handling failed analysis. The first option invalidates the transformation; this option is conservative, but may invalidate legal transformations. The second option inserts runtime monitoring code that raises an exception if the indirect branch targets an invalid destination; this option allows possibly invalid transformations, but may result in unexpected runtime failures. The third option removes all out-edges from the ICT and replaces them with a single edge to the sink node; this option is maximally permissive, but may allow invalid transformations and thus may be dangerous.

3.5 Implementation

We implemented our structured binary editing techniques in the Dyninst 7.0 binary analysis and instrumentation system [11] by extending the PatchAPI component [55] of the system. When compared to other instrumentation systems Dyninst 7.0 has two clear advantages. First, it uses the CFG as its primary program representation, and thus gave us the foundational abstractions on which to build. Second, it provides the analysis capabilities (e.g., slicing) required for our dynamic validity analysis as described in Section 3.4. In this section, we briefly describe the Dyninst 7.0 system and detail how we extended it to become a structured binary editor.

While Dyninst 7.0 uses the CFG as its program representation, this CFG is essentially static; instrumentation is specified as an annotation rather than a transformation, and the augmented CFG is not visible to the user. Thus, this approach does not suffice for structured binary editing. We extended Dyninst 7.0 in three ways: first, by making its CFG user-modifiable; second, by allowing insertion of new code into the CFG; and third, by extending its code generation to instantiate a new binary from the modified CFG.

We extended the PatchAPI component to make the CFG directly modifiable via the transformations described in Section 3.3. One interesting problem was unifying our CFG transformations with Dyninst 7.0's native instrumentation capability, as we wanted to allow a user to use both mechanisms simultaneously. We addressed this problem by introducing a callback mechanism to update instrumentation when the CFG is modified. For example, if a user removed an instrumented block, we will notify them that the instrumentation associated with that block was also removed.

We also extended Dyninst 7.0's internal code-generation mechanism. Dyninst 7.0's instruction-relocation mechanism was limited to regenerating the original CFG with instrumentation added. We extended this mechanism to allow us to redirect edges by creating new branches or altering existing instructions. The result is a buffer of code that instantiates the transformed CFG; we then use Dyninst 7.0 to patch branches from original code to the new code where appropriate (e.g., at corresponding function entry points). One challenge with this approach is handing indirect branches, since we could not simply update an offset in an existing branch. Dyninst 7.0 handles indirect branches by emulating their original behavior, and then patching branches from each original destination to the corresponding location in the instrumented code. We extended this mechanism to instead branch to the target of the modified edge.

Finally, we implemented our dynamic validity checker using the slicing capabilities provided by the DataflowAPI binary analysis component of Dyninst 7.0. This component gives us the ability to efficiently calculate local slices by performing a search over the CFG; this mechanism is inspired by the EEL [36] slicing algorithm. We implemented our analysis by calculating the slice from each indirect jump over the original CFG and then replicating the slice over the transformed CFG. Our current implementation disallows any transformation that will alter a slice. As future work, we plan to instead re-analyze the destinations of the modified jump.

3.6 Case Studies

We conclude this chapter with a presentation of two case studies that demonstrate the efficacy of structured binary editing. The first case study presents a tool for applying security patches to an executing Apache web server without disrupting the server's execution and compares this tool to an approach based on instruction-level modification. The second case study presents ongoing research that uses binary modification to alter the precision of floating-point instructions; this research is being performed by Michael Lam at the University of Maryland [35].

Patching Apache

Hot patching applies changes to a binary program while it is being executed. This is an interesting problem, as it combines aspects of code analysis, code specification, and binary modification. Hot patching a program requires three steps. First, identifying the patch site(s) in the binary from an examination of the source code patch. Second, constructing a new sequence of code that matches the post-patch version of the source code. Third, removing the corresponding original code and replacing it with the new code. Identifying the code to be patched benefits from a representation such as a CFG that closely matches the original code; however, code optimizations, such as function inlining, can complicate identification. Constructing a new code sequence is complicated by the need to refer to, and thus identify, program variables. While it is possible to perform such construction in assembly code, it is often more convenient to use a higher-level language that can refer to program variables, such as the Dyninst 7.0 AST or DynC languages. Removing code may be performed on instructions, the CFG, or functions. Instruction removal requires precisely identifying which instructions should be removed; without some structural information, such as functions, it can be difficult to identify these instructions. In turn, function replacement is frequently too coarse-grained; if functions were inlined in the original binary, it can be difficult to construct a similarly inlined replacement function. CFG modification provides both the structural characteristics that are helpful in identifying the code to replace and the fine-grained control that allows replacement of small sequences of instructions.

Hot patching is complicated by the differences in binaries generated by different compilers, compiler versions, or optimization levels. Ideally, a hot patching tool should be independent of a particular compilation of the code, rather than needing to be specialized for every possible binary. Thus, the method used for code identification, construction, and removal should be related to structural characteristics rather than a particular instruction sequence. For example, the location where local variables are stored may vary, or instructions may be reordered by different compilers.

Current binary modification toolkits are not well-suited to performing hot patching. DynamoRIO [10] and Valgrind [50] can perform instruction-level modification and thus can replace code, but do not provide the structural characteristics helpful for identifying where to apply a patch and require the user to manually construct the replacement code. Furthermore, such identification would be particular to a particular binary since it is instruction based. PIN [41] provides function-level modification, but suffers the same problems as DynamoRIO and Valgrind. Dyninst 7.0 [11] represents the binary as a CFG, thus easing identification, and provides a high-level language for constructing the replacement code sequence. However, like PIN, 7.0 only supports function-level replacement. Our structured binary editing technique addresses this lack by providing CFG modification.

We investigated hot patching security vulnerabilities in a running Apache HTTPD web server [4]. We selected Apache for three reasons. First, it is widely used. Second, security flaws, as well as the patches necessary to fix these flaws, are widely published and available. Third, as a long-running process, Apache is an excellent test of our ability to modify a running process without corrupting its structure. We constructed a single tool that dynamically patches three security vulnerabilities (CVE-2011-3368 [46], CVE-2011-3607 [47], and CVE-2012-0021 [48]) in a running Apache process.

This evaluation has three goals. First, we should be able to patch an unmodified, executing Apache HTTPD web server. Second, we should be able to patch versions of the same server as compiled on different systems, which requires identifying the locations to patch by structural characteristics (e.g., a subgraph of the CFG) rather than a literal sequence of instructions. Third, we should be able to prepare the patch from the corresponding source code patch instead of manually crafting it in assembly code. We accomplished these goals as follows. We prepared our target binaries by compiling the appropriate versions of Apache from source using default configuration options and several different versions of GNU GCC [24]. We used different compiler versions to ensure that our location match did not depend on the particular idioms used by a single version of GCC. Finally, we wrote each patch in Dyninst 7.0's high-level AST language [11] instead of assembly code.

For each security flaw we patched, we began by installing several unpatched versions of Apache compiled with different versions of GCC, from 4.1 to 4.6. For each version, we began by benchmarking the server to verify that it was working properly. Second, we confirmed that the security flaw was present using the appropriate exploit code. Third, we executed our hot-patching tool on the vulnerable web server while it was executing and serving requests. Fourth, we used the exploit tool to verify that the flaw had been successfully patched. Fifth, we benchmarked the server again to verify its correct operation; for each patch, there was no identifiable slowdown in operation (as we would expect, since the patched code was not exercised except by the specific exploit).

For conciseness, we describe how we created our hot patching tool for the CVE-2011-3368 [46] vulnerability. This vulnerability allowed a user to use a carefully crafted URL to gain full internal network access from a DMZ web server. We began by examining the Apache source code and the vulnerability patch to gain an understanding of the vulnerability and to locate where to apply our patch; this code is shown in Figure 3.8.

We then examined the CFG of the binary to identify the corresponding point in the binary that required patching. Interestingly, we found from an examination of the binary that the patched function (`read_request_line`) was not present in the server binary. This function is declared static in its source file, resulting in the compiler inlining it into its caller, `ap_read_request`. However, we were still able to identify its characteristic CFG and prepared the fingerprint shown in Figure 3.9. This fingerprint matched a portion of the `ap_read_request` function that had incorporated `read_request_line`. This fingerprint was unique in the binary; however, this may not be true for binaries built for other compilers. We designed our hot patching tool to exit without modifying the binary if it either fails to find the fingerprint or finds multiple matches, since this would indicate the fingerprint may have falsely identified the location to modify. We performed this step manually; however, we believe it could be partially automated by generating the expected CFG from source code and then manually refining the fingerprint.

Next, we converted the source-code patch file, as shown in Figure 3.10, into a Dyninst 7.0 code snippet. We divided the patch into two sections: a *condition* that identified when server was being attacked and a *body* that took corrective action. Both the condition and body required access to two local variables in `read_request_line`, *r* (a status structure) and *uri* (the input URI). Since Apache is compiled with debugging information, we were able to identify *r* and its fields. However, debugging information for *uri* was not present. However, it is an argument to a function call to `ap_parse_uri` immediately before the patch location. We could not use the argument register directly because it was modified by the call; instead, we used Dyninst 7.0's dataflow analysis capabilities to identify the register that contained the authoritative value for *uri*. Once we had created AST nodes for these two variables, building the snippet was a straightforward operation of manually translating C code. As with the previous step, we generated the code snippet manually. We used the Dyninst AST language to construct the snippet. A simpler alternative would have been to use the DynC [53] C-like language; we did not simply because of lack of familiarity with DynC.

Finally, we constructed a CFG transformation that injected the snippet into the binary; we show the transformed CFG subgraph in Figure 3.11. This

```
1  ap_parse_uri(r, uri);
2  if (ll[0]) {
3    r->assbackwards = 0;
4    pro = ll;
5    len = strlen(ll);
6  } else {
7    r->assbackwards = 1;
8    pro = "HTTP/0.9";
9    len = 8;
10 }
11 r->protocol = apr_pstrmemdup(r->pool, pro, len);
```

Figure 3.8: Code fragment from Apache HTTPD 2.2.21 that surrounds the security fix site; all variables are as named by the Apache developers. The fix is inserted between the the call to `ap_parse_uri` (line 1) and the following if statement (line 2).

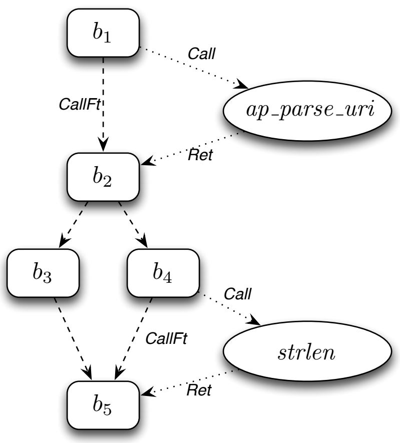


Figure 3.9: CFG fingerprint corresponding to the code fragment shown in Figure 3.8.

```
  ap_parse_uri(r, uri);

+ if (r->method_number != M_CONNECT
+     && !r->parsed_uri.scheme
+     && uri[0] != ’/’
+     && !(uri[0] == ’*’ && uri[1] == ’\0’)) {
+       r->args = NULL;
+       r->hostname = NULL;
+       r->status = HTTP_BAD_REQUEST;
+       r->uri = apr_pstrdup(r->pool, uri);
+ }

  if (ll[0]) {
    r->assbackwards = 0;
    pro = ll;
```

Figure 3.10: Code listing for the CVE-2011-3368 security patch; the lines prefixed with a "+" are inserted. For clarity, this listing omits call to a logging function `ap_log_rerror`.

injection was straightforward, as the patch did not contain complex control flow. We then verified that our tool closed the security flaw while not impacting the normal execution of the web server.

This case study demonstrates that our structured binary editing approach is capable of patching security vulnerabilities in an unmodified, executing Apache web server. We are able to use the same tool on Apache binaries compiled with several different compiler versions. This is due to our use of the CFG to identify the patch site and apply the patch; although the underlying instructions in each compiled version of Apache are different, their CFGs are the same.

In all, our tool consisted of 498 lines of code, of which 93 were responsible for identifying the patch location, 187 were responsible for creating the snippet, and 120 were responsible for transforming the CFG; the remainder was utility code. The small size of our tool is due to two factors: the expressive power of our CFG transformations and the analysis capabilities of the Dyninst 7.0 toolkit.

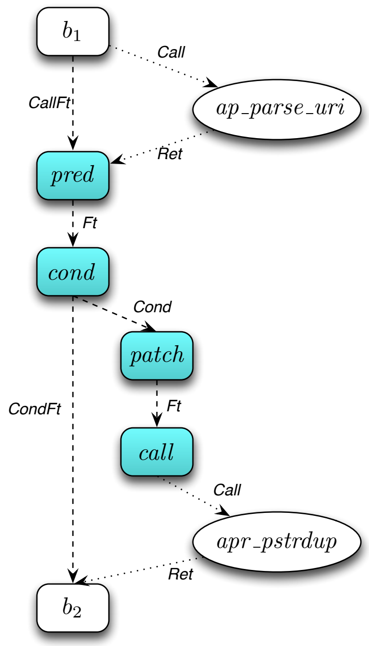


Figure 3.11: The resulting CFG after our transformations were applied to insert the patch shown in Figure 3.10. User-inserted snippets are shown in blue (*pred* and *patch*), and automatically generated code in green (*cond* and *call*). The *pred* snippet corresponds to lines 3-6 and the *patch* snippet to lines 7-10; for clarity, we have omitted a logging call to `ap_log_rerror`. The *cond* block represents the conditional branch used to implement the **if** statement in line 3, and the *call* block represents the call to apr_pstrdup in line 10. We composed three types of transformations to create this CFG: interception insertion, predicate insertion, and call insertion.

We believe these sizes would be representative of other uses of structured binary editing.

Floating Point Precision Modification

Our second case study presents a tool, CRAFT (Configurable Runtime Analysis for Floating-point Tuning), that uses binary modification to alter the precision of floating-point instructions. This work is being performed by Michael Lam at the University of Maryland as part of his doctoral research.

Floating-point operations can be performed at varying levels of precision. Low-precision operations, such as IEEE single precision, have the advantage of significant speed over the double-precision representation. Imprecision in a computation can be dangerous; real-world incidents involving imprecise floating-point computation include the Patriot missile failure in 1995 [32] and the Vancouver stock index slump in the 1970s [27]. However, performing all operations at high precision leads to unnecessary overhead, memory usage, and energy usage. The goal of CRAFT is to automatically determine which operations in a binary must be performed at high precision to maintain a low error rate, and which operations can be performed at low precision instead. CRAFT does this by replacing individual high-precision instructions with low-precision equivalents and determining whether this replacement results in an unacceptable increase in the imprecision of the resulting computation.

CRAFT was designed to use static binary modification instead of source or compile-time modification to allow it to operate on binaries that use vendor-provided math libraries. These libraries are critical to floating-point math performance on recent systems. However, they are typically provided as binaries with minimal debugging information, which prevents them from being included in a compile-time modification step. Instead, they can only be modified at the binary level. Furthermore, CRAFT was designed to use binary rewriting to allow the cost of modification to be amortized over multiple executions of the modified program.

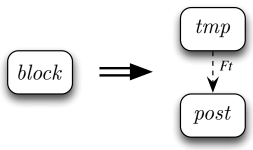
CRAFT uses binary modification to replace individual floating-point instructions with sequences that perform the appropriate floating-point operation in a lower precision, converting the input values to the lower precision if necessary. It tracks the resulting imprecision to determine if the overall computation has unacceptable imprecision.

These operations of instruction replacement and instrumentation are well-supported by instruction-level modification toolkits such as DynamoRIO [10] or Valgrind [50]. However, these tools do not provide binary rewriting. Instead, CRAFT was implemented on a version of Dyninst 7.0 extended with structured binary editing, thus leveraging Dyninst 7.0's binary rewriting capabilities. We show the instruction replacement transformations in Figure 3.12. Replacing an instruction in the middle of a basic block is a straightforward matter of removing the original instruction and using an interception transformation to insert the replacement snippet of code. Instructions at the start of a block are more complicated, since there are multiple in-edges that must be handled.

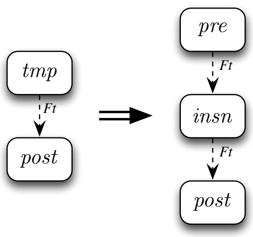
This case study demonstrates that structured binary editing can be used to provide instruction-level binary modification in addition to CFG modification. As with our hot patching tool, such modification can be performed with a small tool. While CRAFT includes 18,000 lines of code, only 3,400 are responsible for modifying the binary.

3.7 Summary

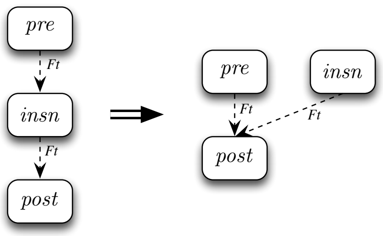
We presented a technique for modifying binaries by transforming their control flow graphs. Our technique, *structured binary editing*, uses an algebra of graph transformations that preserve the validity of the CFG. It also uses a dataflow analysis that determines whether a particular transformation will alter any indirect control flow. We evaluated our technique in two ways. First, we developed a tool for hot patching executing Apache processes; this tool leveraged the capabilities of Dyninst 7.0 to identify where to patch and our algebra for performing the modification itself. Second, we described a tool, CRAFT, developed by Michael Lam that uses structured binary editing to perform instruction-level replacement of floating point instructions. In the next chapter, we extend structured binary editing to operate on functions and loops.



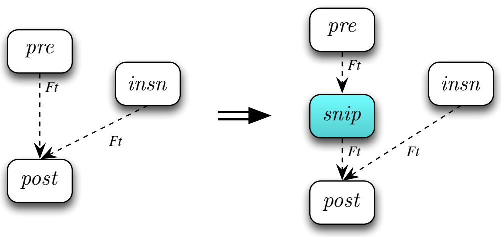
(a) First Block Split



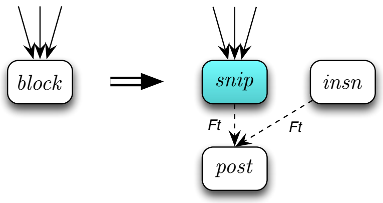
(b) Second Block Split



(c) Edge Redirection



(d) Edge Insertion



(e) First Instruction Replacement

Figure 3.12: Examples of the transformations used by CRAFT. Figures (a) through (d) show replacing an instruction in the middle of the block using block splitting, edge redirection, and edge insertion transformations. Figure (e) shows the result of replacing the first instruction in a block. Instead of the first block split transformation, we instead redirect all in-edges to the replacement snippet.

4 Projections on the CFG: Functions and Loops

In this chapter, we build upon our structured binary editing approach for modifying binaries and extend it to modify the higher-level structural abstractions of functions and loops. These abstractions provide a familiar interface for encapsulating binary complexity. For example, a user can clone a function or insert code at its entry point without understanding any of the details of the underlying code. This approach allows us to handle complexities arising from compiler optimizations or features of the instruction set that cannot be handled by previous function-level instrumentation or modification approaches.

We begin by defining our function abstraction, extending the abstraction used by Dyninst 7.0 [11]. Modern compilers focus on generating efficient binary code instead of preserving the structure of the original source code. As a result, source code functions may be combined, split into multiple segments, or optimized to share code. This frequently makes recapturing the original functions of the source code impossible. Instead, we define an abstraction that is similar at the conceptual level to source code functions but that are derived entirely from the binary.

Next, we discuss modification of functions. We define two classes of function transformations. The first class inserts code at function entry, exits, or call sites; these are familiar points based on the semantic structure of a function. Current binary modification toolkits map these points to underlying instructions, such as mapping function entry to its first instruction [11, 36, 38, 41, 52, 57, 68, 69]. This approach does not correctly handle certain code sequences, such as functions that omit preamble code (e.g., stack frame setup) or that use conditional return instructions. We instead use graph transformations to ensure that code inserted at these points executes precisely as the user expects. The second class of transformations operate on a function as a collection of blocks, and allow users to copy or remove whole functions instead of operating a block at a time.

We then define our loop abstraction which is based on the Dyninst 7.0 loop abstraction [15]. Like functions, loops have well-understood structural characteristics: the entry and exit of the loop as well as the loop back-edge or back-edges. We define transformations to insert code based on these characteristics. As with functions, we also define transformations that act upon the loop as a collection of blocks.

We conclude the chapter with a presentation of experimental results. We compare our approach with other binary modification toolkits, and show that graph transformation-based code insertion can handle code complexities that are not correctly handled by these toolkits.

4.1 Function Definition

The concept of a function is well-defined in source code, and programming languages specify both the semantics and syntax of this construct. For example, functions have well-defined entry points and exit points, and function bodies are contiguous. The same is not true of binaries. Support for function call and return behavior was proposed as early as 1946 by Alan Turing in his Automatic Computing Engine [71], and supported in hardware in the EDSAC machine [73] in 1947. However, no modern architecture mandates the use of call and return instructions to enter and leave functions. For example, the PowerPC architecture does not have a dedicated return instruction; the `blr` instruction that performs a return may also be used as an indirect branch. Similarly, while functions often have code to set up and tear down a stack frame, such code is not required and thus may be eliminated as an optimization. Finally, the debugging information provided by the compiler to help map binary code back to the source code may be missing or (surprisingly often) erroneous. As a result, it is difficult to identify the code regions corresponding to the original source code functions.

Recall that we define our CFG as a graph $G = (V, E, V_e, V_x, T)$, where V is a set of vertices consisting of basic blocks and a unique sink node, E represents the edge set between blocks, V_e and V_x represent the entry and exit blocks, and T is a function that assigns types to edges.

We use the conventional definition of a function as the set of blocks reachable from a single *entry block* traversing only intraprocedural edges. Blocks can be identified as entries by several criteria, including having a call in-edge, being identified in the symbol table as a function entry point, or being identified by a parsing heuristic. When we reach a block that ends in one or more call edges, we instead traverse the corresponding call fallthrough edge if it exists (e.g., if the callee is a returning function). If it does not, we conclude the call is non-returning, such as a call to `exit`, and label the call block as an exit block. Functions do not return if they either halt execution, such as `abort` or `exit`, or call only such functions; thus, there may be chains of non-returning functions in the CFG. If we reach a block with one or more return edges we also label it as an exit block. Similarly, if we identify a tail call we conclude this is an exit point; otherwise, we treat such constructs as jumps. As a result, function bodies may include code that was logically a callee in source code but was optimized to be reached via jumps rather than calls. By using a CFG traversal, we can navigate around data embedded in code, such as a jump table. Similarly, we can identify interleaved function blocks.

Our function model uses a single entry block. This is a common approach in function representations [11, 36, 38, 41, 52, 57, 68, 69], and derives from the common programming language requirement that functions have single entries. However, languages may allow functions to have multiple entry points, such as provided by the Fortran `ENTRY` statement. We represent such functions as a set of single-entry functions that may share code; this approach is inherited from Dyninst 7.0 [11]. This approach represents multiple-entry functions with a minor cost in efficiency. Let M be a function with multiple entries and $S_1, \ldots, S_m$ be the equivalent set of single-entry functions. With the single-entry approach, a dataflow analysis of M can be performed by instead analyzing $S_1, \ldots, S_m$ and combining the results. Similarly, modification of M can be performed by applying an equivalent modification to each $S_1, \ldots, S_m$.

We define functions as subgraphs of the CFG: $f_i = (V_i, E_i, v_i, X_i, T')$ where $V_i \subseteq V$, $E_i \subseteq E$, $v_i \in V_i$ is the *entry block* of the function, $X_i \subseteq V_i$ are the *exit blocks*, and T' assigns types to edges. Unlike the CFG labeling function T, T' can only assign intraprocedural edge types.

Our function abstraction is permissive enough to allow us to represent functions on a wide range of binaries, from conventional to highly optimized. In addition, we handle multiple architectures; while we currently support IA-32/x86-64 and PowerPC, we have supported SPARC and IA-64, and expect that this model can support ARM without further modification as it is similar to PowerPC from a control-flow perspective. Specifically, functions may be noncontiguous in the binary or overlap; entry blocks may have intraprocedural in-edges from other blocks in the function body; and exit blocks may have intraprocedural out-edges to other blocks in the function body.

Functions may share blocks, and we say these functions *overlap*. This is a natural result of our definition of functions as subgraphs of the CFG; if a block is reachable from two different entry blocks traversing only intraprocedural edges then it is shared. Clearly, if a block is shared then all of its successor blocks are also shared until an exit block is reached. Modern compilers frequently share common code between functions, such as register restores or error code, to increase code density; we show an example in Figure 4.1.

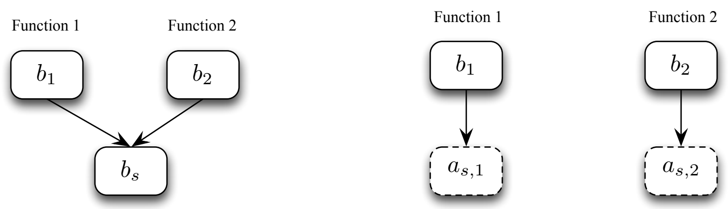
Overlapping functions complicate modification because users frequently expect functions to be independent; that is, a modification of a block in one function should not affect other functions. This assumption is broken by overlapping functions. We handle this as follows. We represent shared blocks in our interprocedural CFG (Figure 4.2a), but hide it at the function layer; as a result, overlapping functions can be treated as disjoint. We do this with *block aliases* (Figure 4.2b). A block alias represents a shared block in the context of one particular function, and any analysis or instrumentation performed on the alias is restricted to that function. More formally, let b_s be a block shared by functions $f_1, \ldots, f_n$. For each such function f_i we create a *block alias*, denoted $a_{s,i}$. Block aliases replace their corresponding blocks in each function's CFG. We maintain a link between the underlying block and its block

```
<memcpy_chk>:
  63bb40:        mov    0xc(%esp),%eax
  63bb44:        cmp    %eax,0x10(%esp)
  63bb48:        jb     6b0930 <__chk_fail>

<stpcpy_chk>:
  6afcb0:        push   %ebp
  6afcb1:        mov    %esp,%ebp
  6afcb3:        mov    0x8(%ebp),%ecx
  6afcb6:        push   %esi
  6afcb7:        mov    0xc(%ebp),%edx
  6afcba:        mov    0x10(%ebp),%esi
  6afcbd:        jmp    6afcc3 <__stpcpy_chk+0x13>
                . . .
  6afcc3:        sub    $0x1,%esi
  6afcc6:        cmp    $0xffffffff,%esi
  6afcc9:        je     6afcde <__stpcpy_chk+0x2e>
                . . .
  6afcde:        call   6b0930 <__chk_fail>

<__chk_fail>:
  6b0930:        push   %ebp
  6b0931:        mov    %esp,%ebp
  6b0933:        push   %edi
  6b0934:        push   %esi
  6b0935:        push   %ebx
  6b0936:        call   5dfce0 <__i686.get_pc_thunk.bx>
  6b093b:        add    $0x6e6b9,%ebx
  6b0941:        sub    $0xc,%esp
  6b0944:        lea    0xfffd21e9(%ebx),%edi
  6b094a:        lea    0xfffd46cc(%ebx),%esi
  6b0950:        mov    0x31f0(%ebx),%eax
  6b0956:        mov    (%eax),%eax
  6b0958:        mov    %esi,0x4(%esp)
  6b095c:        movl   $0x2,(%esp)
  6b0963:        test   %eax,%eax
  6b0965:        cmove  %edi,%eax
  6b0968:        mov    %eax,0x8(%esp)
  6b096c:        call   62b190 <__libc_message>
  6b0971:        jmp    6b0950 <__chk_fail+0x20>
```

Figure 4.1: Example of two functions sharing code taken from GNU libc on IA-32 compiled with GNU GCC 4.x. Both `memcpy_chk` and `stpcpy_chk` share code with the `__chk_fail` function; this code is also shared by several other libc functions. Both of these functions take an extra parameter representing the size of the destination buffer, and abort via the shared code if the destination is smaller than the source.



(a) Interprocedural CFG (b) Block Aliases

Figure 4.2: An example of functions sharing blocks. Figure (a) shows the interprocedural CFG, with two functions sharing the same block b_s. Figure (b) shows the corresponding function representations, with the return block represented by two aliases, $a_{s,1}$ and $a_{s,2}$.

aliases; therefore, a user can determine if a block belongs to multiple functions by checking for the existence of block aliases.

We allow the entry blocks of functions to have intraprocedural in-edges as well as interprocedural in-edges, such as shown in Figure 4.3c. A common cause of such functions is the compiler omitting preamble code. This situation complicates modification, as inserting code at the entry of a function cannot be performed by inserting code at the entry of the entry block.

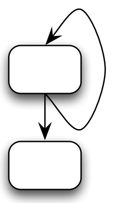
Similarly, an exit block may have an intraprocedural out-edge in addition to its interprocedural out-edges, such as shown in Figure 4.4c. This is caused by architectures that support conditional return instructions, such as PowerPC and ARM. An example of such instructions is the PowerPC `bclr` instruction (branch conditional to link register), which performs a conditional return. PowerPC also supports conditional calls through the `bcctrl` instruction (branch conditional to count register, linking). The ARM architecture provides conditional execution of all instructions by allowing the user to specify an optional predicate. Thus, both the ARM return instruction (`mov pc, lr`) and call instruction (`bl`) may be emitted in a conditional form.

```
int main(int argc) {
  do {
    argc -= 1;
  } while (argc > 0);
  return argc;
}
```

(a) Code Listing

```
400450: sub    $0x1, %edi
400453: test   %edi, %edi
400455: jg     400450
400457: mov    %edi, %eax
400459: ret
```

(b) Disassembly



(c) Function CFG

Figure 4.3: Code listing, disassembly, and CFG for an example function whose first instruction executes multiple times per function invocation. This example was generated by GNU GCC version 4.1.2 at optimization level O2.

4.2 Function Modification

In this section, we extend our graph transformation algebra to operate on functions. We describe two classes of function transformations. The first class inserts code into functions at semantically interesting points in execution: entry, exit, before calls, and after calls. The second class operates on functions as collections of basic blocks: function removal, function cloning, and function unsharing. The unsharing transformation is a refinement of function cloning that clones basic blocks to ensure that the transformed function does not share code with another function.

We define four types of code insertion transformations on functions: function entry, function exit, pre-call, and post-call. These transformations are shown in Figures 4.5 and 4.6. Function entry insertion adds code that is executed immediately before the body of a function executes. If the entry block has no intraprocedural in-edges, this is logically equivalent to adding code at the start of the block; if such in-edges exist, we transform the function to have

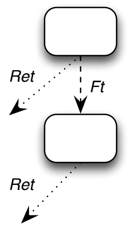
```
int i;

void bar(int j) {
  if (j < 5) return;
  i = j + 1;
  return;
}
```

(a) Code Listing

```
<bar>:
100004a0: cmpwi   cr7,r3,4
100004a4: blelr   cr7
100004a8: lis     r9,4097
100004ac: addi    r0,r3,1
100004b0: stw     r0,4108(r9)
100004b4: blr
```

(b) Disassembly

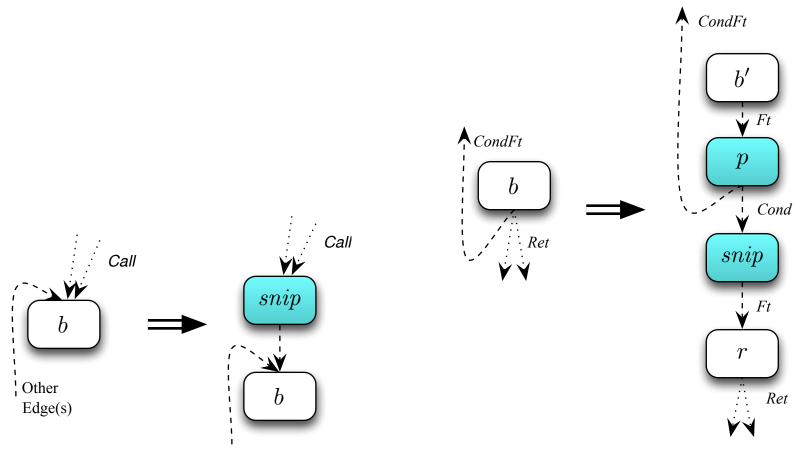


(c) Function CFG

Figure 4.4: Code listing, disassembly, and CFG for an example function with a conditional return instruction. The function first compares its input parameter (in `r3`) to 4 and stores the result in condition register 7 (`cr7`). If the parameter is less than or equal to 4 it returns immediately via a `bclr`, written as `blelr` to encode the comparison. Otherwise, writes the global variable via a load (`lis`), add (`addi`), and store (`stw`) before returning (`blr`). This example was generated by GNU GCC version 4.3.2 at optimization level O2.

a distinct entry block. This approach is similar to CFG representations that use a virtual entry block to ensure that such entries have no intraprocedural in-edges, such as done by PLTO [63]. We insert code post-call similarly, so it is executed immediately after the called function returns and before any other function code executes.

Exit and pre-call code insertion is more complex. These points are characterized by a control transfer instruction. For example, a function exit point normally corresponds to a return instruction, although other control transfers might be used, such as a jump in a tail call or a call to a non-returning function. Binary modification toolkits conventionally insert code immediately

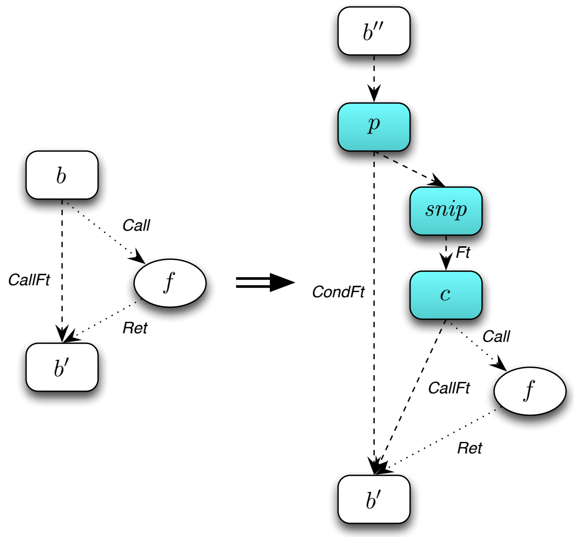


(a) Function Entry Transformation (b) Function Exit Transformation

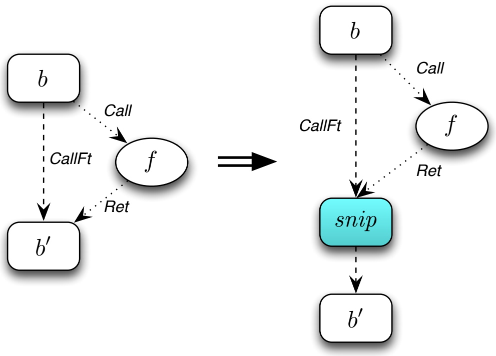
Figure 4.5: CFG transformations for function entry and exit code insertion; in each case, the inserted code is represented by a blue-shaded block $snip$. Function entry instrumentation is performed by redirecting all interprocedural edges to $snip$, while intraprocedural edges remain unaffected. Function exit instrumentation is complicated by conditional return instructions, such as supported by the PowerPC architecture. We first split the exit block b into two blocks b' and r, where r contains only the return instruction. The blue-shaded block p represents a predicate equivalent to that used by the return instruction.

before such instructions execute so that the new code executes in the context of the transformed function instead of its caller (for code inserted at exit) or callee (for code inserted pre-call) [11, 36, 38, 41, 52, 57, 68, 69]; we use this convention. However, these tools unconditionally insert such code, and thus will erroneously execute the inserted code if the call or return is conditional. Our design ensures that the inserted code executes only if the call or return is taken.

The second class of transformations operates on functions as collections of blocks. Function removal destroys a function f and blocks that belong only to f; any shared blocks are not removed. For removal to be valid, the entry block v_f of f may not have any in-edges from outside the body of f, since such in-edges are not removed. We define this algorithm in Figure 4.7. Loop removal operates similarly; as with function removal, the entry block of the



(a) Pre-Call Transformation



(b) Post-Call Transformation

Figure 4.6: CFG transformations for pre-call and post-call code insertion; in each case, the inserted code is represented by a blue-shaded block $snip$. Pre-call instrumentation is performed by splitting the block b into two blocks b'' and c, where c only contains the call instruction. We introduce a predicate snippet p before the inserted snippet. Post-call instrumentation is simpler; we simply interpose the new code along the call fallthrough and return edges.

```
Algorithm: Function Removal
input  : A graph $CFG$ and function $f = (V_f, E_f, v_f, X_f, T_f, L_f)$
output : A modified graph $CFG'$ with any non-shared blocks in $V_f$ removed
$CFG' \leftarrow CFG$, $ToDel \leftarrow \emptyset$;
for each block $b \in V_f$ do
    if $b$ is not shared and $b \neq v_\perp$ then
        $ToDel \leftarrow ToDel \cup \{b\}$;
        for each in-edge $e = (b', b) \in In(b)$ do
            EdgeRedirect($e, v_\perp, CFG'$);
        end for
    end if
end for
for each block $b \in ToDel$ do
    BlockRemove($b, CFG'$);
end for
```

Figure 4.7: An algorithm for removing a function f and all of its non-shared blocks. The entry block v_f of f must have no interprocedural in-edges; intraprocedural in-edges will be deleted as part of the transformation. We begin by redirecting all in-edges to non-shared blocks to the sink node; this ensures that all blocks we wish to remove will have no in-edges remaining (lines 3-10). We then iterate and remove all such blocks (lines 11-13).

loop cannot have any in-edges from outside the loop body.

Function cloning creates a copy f' of a function f by cloning all blocks in f and redirecting the out-edges of these cloned blocks appropriately. Note that the entry block v'_f of f' will have no interprocedural in-edges (as it was just created and thus is not reachable from outside of f') but may have intraprocedural in-edges from other blocks in f'. We define an algorithm for function cloning in Figure 4.8, and show an example in Figure 4.10. Loop cloning operates similarly, creating copies of all blocks in the loop body.

Function unsharing eliminates sharing of blocks between functions. We do this by cloning all shared basic blocks, with the exception of the sink node. This algorithm is similar to function cloning, but has two key differences. First, function unsharing only copies shared blocks rather than all blocks. Second, it does not create a new function, instead copying blocks within the original. We define this algorithm in Figure 4.9.

```
Algorithm: Function Cloning
input  : A graph $CFG$ and function $f = (V_f, E_f, v_f, X_f, T_f, L_f)$
output : A modified graph $CFG'$ and new function $f'$
$CFG' \leftarrow CFG$, $V'_f \leftarrow \emptyset$, $E'_f \leftarrow \emptyset$, $v'_f \leftarrow v_\perp$, $X'_f \leftarrow \emptyset$;
for each block $b \in V_f$ do
    if $b \neq v_\perp$ then
        $b' \leftarrow$ BlockClone($b, CFG'$);
    else
        $b' \leftarrow v_\perp$;
    end if
    $V'_f \leftarrow V'_f \cup \{b'\}$;
    if $b = v_f$ then
        $v'_f \leftarrow b'$;
    end if
    if $b \in X_f$ then
        $X'_f \leftarrow X'_f \cup \{b'\}$;
    end if
end for
for each block $b' \in V'_f$ do
    for each out-edge $e' = (b', b_t) \in Out(b')$ do
        $E'_f \leftarrow E'_f \cup \{e'\}$;
        if $e'$ is intraprocedural then
            $b'_t \leftarrow$ the clone of $b_t$;
            EdgeRedirect($e', b'_t, CFG'$);
        end if
    end for
end for
```

Figure 4.8: An algorithm for cloning a function f into a function f'. As with block cloning, the entry of f' has no interprocedural in-edges. This algorithm constructs the blocks, edges, entry block, and exit blocks of f'; the construction of the type and context labeling functions are omitted. We begin by cloning each block $b \in V_f$ and constructing V'_f, v'_f and X'_f (lines 3-16). After each block has been cloned, we iterate over each and redirect all intraprocedural out-edges to the corresponding cloned blocks (lines 17-25).

4.3 Loops

We also support transformations of loops. We use Dyninst 7.0's definition of a loop [15], which is an extension of natural loops [1]. Loops are defined

```
1 Algorithm: Function Unsharing
  input  : A graph $CFG$ and function $f = (V_f, E_f, v_f, X_f, T_f, L_f)$
  output : A modified graph $CFG'$ such that $f$ shares no blocks with any other
           function
2 $CFG' \leftarrow CFG$, $Tmp \leftarrow \emptyset$;
3 for each block $b \in V_f$ do
4     if $b$ is shared then
5         $b' \leftarrow$ BlockClone$(b, CFG')$;
6     else
7         $b' \leftarrow b$;
8     end if
9     $Tmp \leftarrow Tmp \cup \{b'\}$;
10 end for
11 for each block $b' \in Tmp$ do
12     for each out-edge $e = (b', b_t) \in Out(b')$ do
13         if $e$ is intraprocedural then
14             $b'_t \leftarrow$ the clone of $b_t$;
15             EdgeRedirect$(e, b_t, b'_t)$;
16         end if
17     end for
18 end for
```

Figure 4.9: An algorithm for cloning all shared blocks in a function f. It is often convenient to operate on all blocks in a function in isolation, assuming they are not shared by other functions; this transformation algorithm can enforce this assumption. Unlike the function cloning algorithm, this does not create a new function f'. We begin by cloning all shared blocks (lines 3-10). We then redirect all edges that originally targeted a shared block to its clone, leaving edges that target non-shared blocks unmodified (lines 11-18).

similarly to functions, as subgraphs of the CFG: $l_j = (V_j, E_j, v_j, X_j, T')$ where $V_j \subseteq V$, $e_j \subseteq E$, v_j is the unique entry node, X_j are exit nodes, and T' assigns types to edges. Like functions, T' assigns only intraprocedural edge types. A natural loop is defined by its back edges, and we identify these edges with a depth-first search using Tarjan's algorithm [39]. An edge is a back edge if its source is within the body of the loop and its target is the loop head; that is, an edge $e = (v, v_j)$ is a back edge if $v \in V_j$. Loops may be nested and share entry blocks. However, the combination of entry block, exit blocks, and back-edges uniquely identifies a loop.

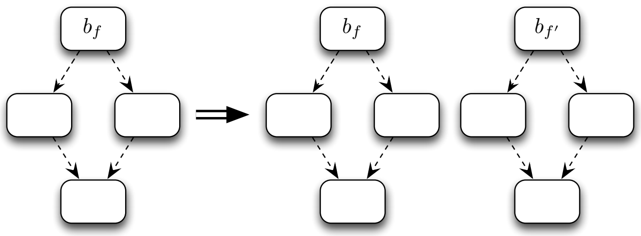
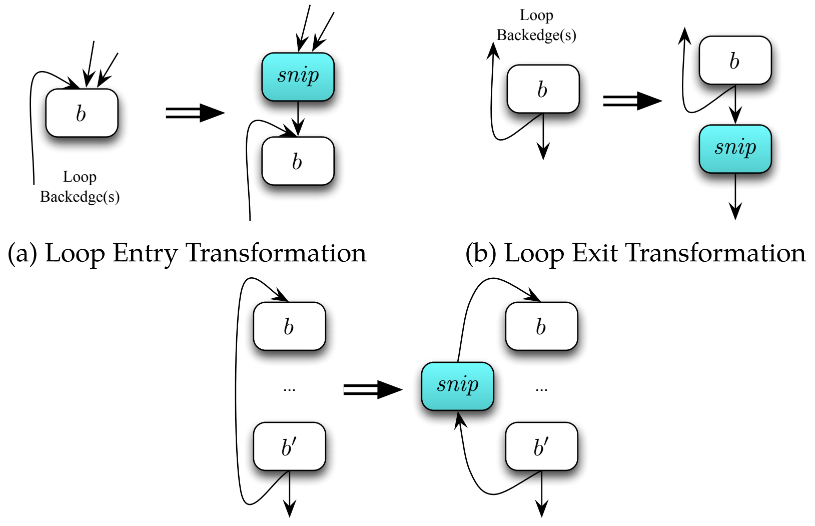


Figure 4.10: Example of function cloning; we copy the function with entry block b_f and consisting of four blocks to a new function with entry block $b_{f'}$.



(a) Loop Entry Transformation (b) Loop Exit Transformation

(c) Loop Iteration Transformation

Figure 4.11: CFG transformations for our loop code insertion transformations. These transformations insert a snippet ($snip$) along the in-edges, out-edges, or back-edges of loops.

	SBE	Dyninst 7.0	PIN	PEBIL
Overlapping Functions	Yes	Yes	No	No
Omitted Preambles	Yes	No	No	No
Conditional Return	Yes	No	-	-

Table 4.1: Summary of results comparing our structured binary editing (SBE) function transformations to Dyninst 7.0, PIN, and PEBIL. A yes entry indicates that the toolkit successfully instrumented that case; a no entry indicates that the toolkit failed. A dash indicates that the experiment was not performed with that particular toolkit.

We define code insertion transformations for loop entry, loop exit, and per-iteration. Like Dyninst 7.0 [15], these transformations are based on inserting this code along edges. We show these loop transformations in Figure 4.11. Since loops are also collections of blocks, we can clone or remove loops by operating on their component blocks in the same way as we do for functions.

4.4 Experimental Results

We conclude the chapter with three experiments that test the effectiveness of our function transformations. We verified that these transformations can correctly insert code in functions that share code, have omitted preambles, or conditional returns. We also performed these tests on the PIN [41], Dyninst 7.0 [11], and PEBIL [38] toolkits, in each case using the function instrumentation interface provided by these toolkits. We did not include SecondWrite [52], DynamoRIO [10], or Valgrind [50] in our experiments. SecondWrite is not publicly available, and neither DynamoRIO nor Valgrind provide a function abstraction. The summary of these experiments is shown in Table 4.1. The first three experiments were performed on a 2.27 GHz Intel quad-core Xeon machine with 6GB of memory, installed with Red Hat Enterprise Server 5.7. The fourth experiment was performed on a 1.2 GHz PowerPC machine with 2GB of memory, installed with OpenSUSE 11.1.

Our first experiment performed function entry and exit instrumentation of overlapping functions (Figure 4.1). Each toolkit we tested provides a single-entry function model. We thus instrumented both the entry and exit of `memcpy_chk` and `stpcpy_chk` and executed each function in succession. We consider a toolkit to have succeeded if the entry and exit output correspond, and failed if the wrong exit instrumentation was executed. Both Dyninst 7.0 and our approach succeeded, due to their explicit handling of overlapping functions in their CFG definitions. PEBIL marked one function as uninstrumentable and so failed this experiment. PIN failed to recognize either function and so did not instrument them.

Our second experiment performed function entry instrumentation of a function with an omitted preamble (Figure 4.3). This code is representative of the optimized code found in larger programs. We instrumented the entry of the function, and consider an toolkit to have succeeded if the entry instrumentation executed once. Dyninst 7.0, PIN, and PEBIL each incorrectly mapped function entry instrumentation to instrumentation of the first instruction, and thus failed this test since the entry instrumentation executed multiple times. Our approach constructed a new entry block that only executed once and thus passed.

Our third experiment performed exit instrumentation of a binary with conditional returns. Unlike the previous experiments, we performed this experiment on a PowerPC machine since that architecture supports these instructions (as `bclr`), using the binary shown in Figure 4.4. We considered the toolkit to have succeeded if the exit instrumentation only executes when the corresponding instruction was taken. We included only Dyninst 7.0 and our approach in this experiment, as the other toolkits do not support PowerPC. Dyninst 7.0 unconditionally executed instrumentation before the return instructions, failing the experiment. Our approach correctly executed instrumentation conditionally.

4.5 Summary

We extended our structured binary editing technique to operate on functions and loops as well as the CFG. Functions and loops represent code structures with well-understood semantic structures that are derived from programming languages; however, their definitions in terms of binary code are complicated by compiler optimizations and architectural features. We define an efficient, single-entry abstraction of a function that can be applied to highly optimized code across a variety of architectures, and reuse the Dyninst 7.0 loop abstraction [15]. We then defined code transformations for functions and loops. Our experimental results demonstrated that these transformations can successfully insert code in functions where other binary modification toolkits fail. In the next chapter, we briefly digress from binary modification to discuss how we use our graph transformation approach to provide simplified, annotated *views* of a CFG.

5 Instrumentation and Other Views on the CFG

An interesting and important application of binary modification is binary *instrumentation*, which inserts code that adds functionality to the program but has no semantic effect on the original code. Instrumentation is used for several purposes, including performance monitoring [34, 66], debugging [20], attack detection [51], cyberforensics [45], and behavior monitoring [21]. Since instrumentation has no effect on the semantics of the original program, tool builders internally represent inserted instrumentation as an annotation of their program representation [10, 11, 22, 36, 38, 41, 50, 57, 68], and may either expose this annotation to the user [11, 36, 57, 68] or hide it from user view [10, 22, 38, 41, 50].

The transformations required to insert instrumentation can be naturally expressed with our structured binary editing algebra. However, while these transformations provide the capability to insert instrumentation, they cause visible changes to the structure of the CFG. This conflicts with the desires of tool builders and users to not have the presence of instrumentation reflected in the CFG. Our approach to this is to provide *views* of the CFG that allow tool builders present a simplified CFG to the users while internally maintaining the full, transformed CFG. In this chapter, we define a view and the associated concept of an *annotated CFG* that adds additional, non-structural information to the underlying CFG.

Interestingly, the concept of a view can be applied to other uses besides instrumentation. For example, we can represent the block aliases defined in Section 4.1 as a view on the CFG and thus hide shared code from the user.

Similarly, we can use a view to provide alternate representations of the effects of predicated instructions, such as those supported by ARM, on the CFG. We describe these views below.

5.1 Definitions

We begin with the concept of an annotated CFG. CFG annotation allows users to associate labels with vertices or edges in the CFG, such as labeling a block as instrumentation. The CFG is a natural target for annotation, and previous annotations have included the results of dataflow analysis or other semantic information [1]. We could represent such annotations by either extending our block and edge definitions to include this information or using an additional set of mapping functions that associate annotation labels with blocks and edges. As a practical concern, and to better match the application of this concept to existing tools, we use the mapping approach.

We first define an *annotation* on the CFG as a function $A : (V \cup E) \rightarrow L$, where V is the vertex set of the CFG (including the sink), E are edges, and L is a set of annotation-specific labels. We can apply a set of annotations to a CFG; this is denoted as a set $\mathcal{A} = (A_1, \ldots, A_n)$ of annotations to a set $\mathcal{L} = (L_1, \ldots, L_n)$ of labels.

We define an annotated CFG, or *ACFG*, as a tuple $ACFG = (CFG, \mathcal{A})$ where CFG is a CFG as defined in Section 3.1 and $\mathcal{A}$ is a set of annotations as above. Trivially, an unannotated CFG can be treated as an annotated CFG with an empty annotation set.

We define a *view* of an annotated CFG as two components: a *view definition* and a *view application*. A view definition is a tuple $VD = (T_V, \mathcal{A}_V)$ where $T_V : CFG \rightarrow CFG$ is a graph transformation that maps from the input CFG to the viewed CFG and $\mathcal{A}_V$ is an annotation set that the view adds to the resulting CFG. A *view application* is a function $VA : ACFG \times VD \rightarrow ACFG$ applies a view definition to an input ACFG.

Given an input $ACFG = (CFG, \mathcal{A})$ and a view $V = (T_V, \mathcal{A}_V)$, we compute the resulting $ACFG' = (CFG', \mathcal{A}')$ as follows. We determine CFG' by enumerating all subgraphs of CFG that are isomorphic to the left hand side of T_V

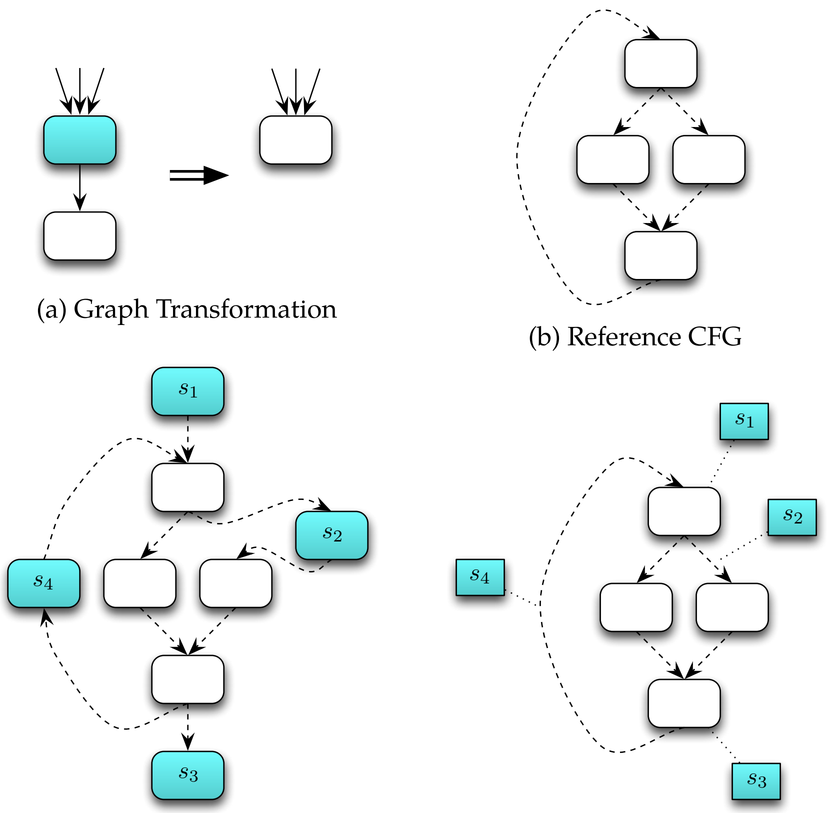
and applying the transformation to each such subgraph. We require that these subgraphs are disjoint and thus there is a unique way to apply T_V to CFG; if not, the results of applying the view are undefined. We merge the annotations associated with the input graph with the annotations specified by the view: $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}_V$.

5.2 Instrumentation and Other Views

The instrumentation view hides the presence of instrumentation code in a CFG, taking advantage of the *net zero* characteristics of instrumentation. Unlike other binary modification, executing a region of instrumentation code will have no cumulative effect on the surrounding original code or data. This characteristic has two important consequences. First, a region of instrumentation must have a single entry and single exit, which is a natural consequence of it not altering the control flow of the program. Second, it cannot cause a change in indirect control flow, since doing so would similarly effect the execution of the program.

For simplicity, we assume that all blocks corresponding to instrumentation are labeled in the CFG; doing so is straightforward when the instrumentation is first inserted. We generate the instrumentation view by applying the graph transformation shown in Figure 5.1a; this transformation removes instrumentation blocks from the CFG. The annotations associated with this view consist of a representation of the removed instrumentation. We show an example of the instrumentation view as applied to a CFG in Figure 5.1.

Interestingly, the CFG resulting from the instrumentation view may not be identical to the original program CFG. For example, if we split a block to insert instrumentation, then the block will remain split in the instrumentation view. Similarly, if we clone a block, such as to provide function-specific instrumentation of a shared block, those blocks will remain cloned in the instrumentation view. However, the resulting CFG will still be valid, if non-minimal, for the original program. Reversing the transformations to split or clone blocks would require labeling such blocks as well as instrumentation blocks. While our current implementation does not do this for reasons of memory efficiency, it



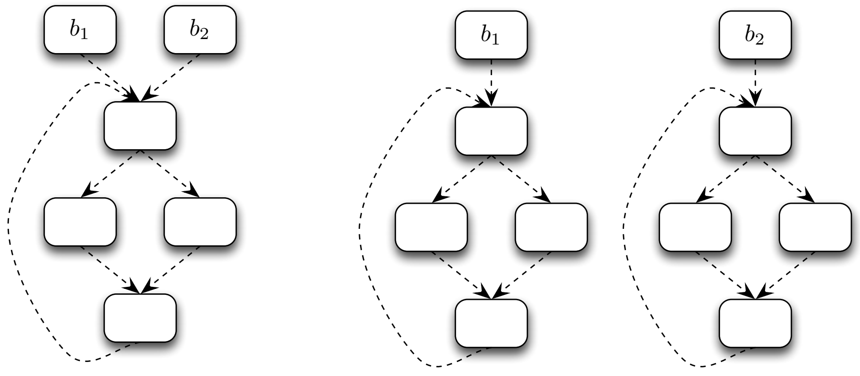
(a) Graph Transformation

(b) Reference CFG

(c) Modified CFG

(d) Instrumentation View

Figure 5.1: Example of our instrumentation view on the CFG. Figure a) shows the graph transformation associated with this view; this transformation removes a region of instrumentation from the CFG. For simplicity, we represent this region as a single shaded block; however, it could be any arbitrary single-entry, single-exit subgraph of the CFG that contains only instrumentation code. Figure b) shows an original CFG for reference. Figure c) shows a modified CFG with instrumentation code inserted; for this example, function entry (s_1), edge (s_2), function exit (s_3), and loop iteration (s_4). Figure d) shows the instrumentation view of the modified CFG, with instrumentation represented as an annotation (blue boxes).



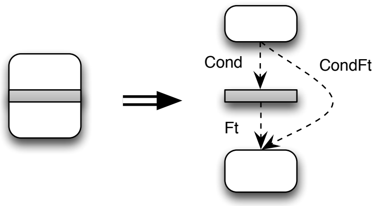
(a) Original CFG (b) Unshared View

Figure 5.2: Example of our function unsharing view on the CFG. Figure a) shows the original CFG, with two entry blocks labeled b_1 and b_2 that result in two overlapping functions. Figure b) shows the unsharing view on the CFG, with the functions entered at b_1 and b_2 having distinct bodies.

would be straightforward to do.

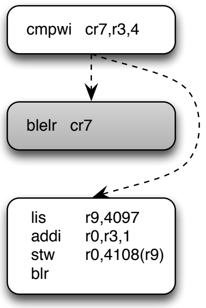
The non-overlapping view simplifies overlapping functions by duplicating all shared blocks so that no functions appear to share code. This view supports the CFG representation used by Dyninst 7.0 [11]. It is implemented with the function unsharing graph transformation defined in Figure 4.9; we show an example in Figure 5.2. Unlike the previous instrumentation view, which removes blocks from the CFG, this view adds additional blocks; these additional blocks serve the same purpose as the block aliases defined in Section 4.1. The annotation function for this view labels each cloned block with its original source block.

Our third view example provides an alternative interpretation of predicated instructions. Such instructions are executed unconditionally, but only have an effect on process state if the encoded predicate (a function of the status flags) evaluates to true. The PowerPC and IA-32/x86-64 architectures support a small number of predicated instructions, including the PowerPC conditional call and return instructions and the IA-32/x86-64 conditional move instruction. The ARM and IA-64 architectures support predication of the majority of their instructions.



(a) Graph Transformation

```
<bar>:
  100005a0: cmpwi    cr7,r3,4
  100005a4: blelr    cr7
  100005a8: lis      r9,4097
  100005ac: addi     r0,r3,1
  100005b0: stw      r0,4108(r9)
  100005b4: blr
```



(b) PowerPC Code Fragment (c) CFG View for PowerPC Example

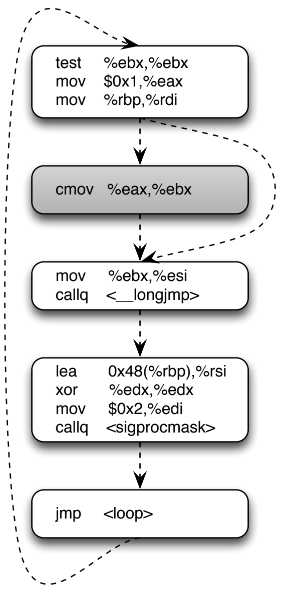
Figure 5.3: Example of our predicate CFG view. Figure a) shows the graph transformation associated with this view; this transformation splits a block around a predicated instruction. Figure b) shows an example of PowerPC code that contains a conditional return (`blelr`) instruction, and Figure c) shows the CFG view associated with this code fragment. For clarity, we only show intraprocedural edges. Note that this view does not annotate the CFG; instead, it splits blocks in the input ACFG that contain predicated instructions. We provide an additional example in Figure 5.4.

There are two ways to consider the effects of such instructions on the CFG: as implicit or explicit control flow. The implicit approach considers the control flow caused by the predicate to be internal to the instruction, and thus does not represent the effects of the predicate in the graph. As a result, predicated instructions do not split basic blocks. This approach reduces the size of the CFG, and is the approach used by the parsing algorithm referenced in Section 3.1. However, this approach requires any user of the CFG to recognize and appropriately handle predicated instructions.

The explicit approach makes the control flow introduced by predicates

```
<__libc_longjmp>:
  ...
loop:
  32118300cf: test    %ebx,%ebx
  32118300d1: mov     $0x1,%eax
  32118300d6: mov     %rbp,%rdi
  32118300d9: cmov    %eax,%ebx
  32118300dc: mov     %ebx,%esi
  32118300de: callq   <__longjmp>
  32118300e3: lea     0x48(%rbp),%rsi
  32118300e7: xor     %edx,%edx
  32118300e9: mov     $0x2,%edi
  32118300ee: callq   <sigprocmask>
  32118300f3: jmp     <loop>
```

(a) x86-64 Code Fragment



(b) CFG View

Figure 5.4: Example of our predicate CFG view, continued. Figure a) shows a more complex example of x86-64 code that contains a conditional move (`cmov`) instruction; this example was taken from the internal GNU libc `longjmp` function. Figure b) shows the CFG view associated with the x86-64 code fragment. For clarity, we only show intraprocedural edges.

obvious in the CFG. Two techniques have been developed to do this: block splitting and predicate bundling. The block splitting technique splits blocks around each predicated instruction. This approach results in a larger CFG with more blocks, but maintains the original instruction order; since we wish to represent the original code, we use this approach in our predicate view. We show our predicate view transformation in Figure 5.3a, along with two examples of its application in Figures 5.3 and 5.4.

The predicate bundling technique analyzes the program to determine which instructions will execute together since they rely on the same predicate, and then constructs basic blocks from these sequences. An example of this technique is the ILTO whole-program binary optimizer [67]. While this technique results in a more compact CFG than the block splitting technique, the instruction order in this CFG may differ from the original program and thus the CFG may diverge from the underlying code. Since we wish to represent the original code as closely as possible, we do not use this technique.

5.3 Summary

We applied our graph transformation approach to providing simplified views of the CFG and extended the CFG to provide annotation capabilities. We briefly introduced these concepts and provided three examples of views on the CFG: an instrumentation view, a function unsharing view, and a predicated instruction view. In the next chapter, we return to binary modification and discuss our patch-based code replacement technique that we use to incorporate transformed code into the binary.

6 Anytime Code Replacement

In Chapters 3 and 4, we discussed the techniques we use to modify a binary in terms of modifying its CFG. In this chapter, we describe the underlying techniques we use to generate new binary code from the modified CFG and replace the original binary code with this new, modified code. Our goals are to replace code at *any time* during execution while imposing *proportional cost*. A binary modification toolkit that provides any time code replacement can modify a program at any point during the execution continuum, including:

- Pre-execution, by modifying the binary on disk via binary rewriting. Binary rewriting allows the user to amortize the cost of producing a modified binary over multiple executions.
- During execution, but executing code that will not be modified. Dynamic instrumentation of such code is similar to binary rewriting since there is no active process state corresponding to the modified region of code.
- During execution, and executing within the modified region. In this case, any modification should take effect immediately to avoid indeterminate delays before the effects of modification are felt by the process. For example, if a user instruments immediately after the currently executing instruction, such instrumentation should also execute immediately.

A toolkit imposes proportional cost if two things hold. First, the size increase of the binary is proportional to the amount of code modified plus the size of any new code. Second, execution overhead is only incurred when executing modified code and not original, unmodified code.

Our anytime code replacement approach extends previous work in patch-based code replacement [11, 36, 38, 68]. We selected patch-based code replacement because it provides proportional cost and operates on arbitrary binaries. In contrast, JIT-based code replacement imposes fixed overhead due to its incremental parsing and code relocation requirement, as discussed in Section 2.3, and in-place code replacement requires specially prepared binaries, as discussed in Section 2.3.

Previous research into patch-based code replacement developed techniques to support both binary rewriting and dynamic instrumentation of arbitrary binaries while imposing proportional cost. However, these patch-based techniques have three weaknesses. First, they assume a one-to-one mapping between basic blocks in the original code and basic blocks in the modified code; this may not hold if the user removed original basic blocks or inserted additional blocks. Second, they treat each basic block or function independently instead of treating all modified blocks or functions as a global unit, and therefore may miss opportunities to reduce overhead by reducing the number of patched interception branches. Third, they cannot ensure that the replacement of actively executing code takes effect immediately.

We describe two novel techniques that address these problems. The first, *region patching*, can replace a region of code, from a single basic block to a set of functions, as a single unit. This technique allows us to both correctly map between the original and replacement regions when modification of the CFG has been performed and reduce the number of interception branches inserted. The second, *state interception*, provides the ability to replace code that is being actively executed.

We conclude the chapter with a comparison of the overhead imposed by anytime code replacement with the techniques used by other toolkits, and show that we impose competitive or lower overall cost. We demonstrate that anytime code replacement imposes proportional cost, and compare this to the non-proportional cost imposed by a toolkit using JIT code replacement. Finally, we show that anytime code replacement can instrument actively executing code that other techniques fail to instrument correctly.

6.1 Region Patching

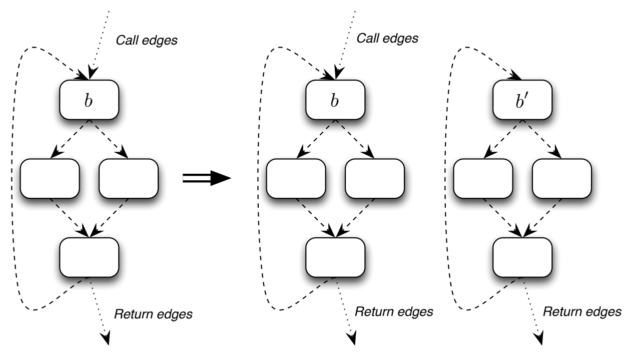
The goal of code replacement is to replace the execution of original code with the user-modified version. Patch-based code replacement does this by copying a region of original code, modifying the copy, and appending the modified copy to the end of the binary; the original code is then overwritten (or *patched*) with *interception branches* to the modified copy. Thus, when the original code would have been executed by the program, the program will instead branch to and execute the modified code. We rely on a technique called *code relocation*, which we describe in the next chapter, to allow us to construct this modified region at the end of the binary, instead of in-place, without changing its behavior.

We describe patch-based code replacement as a further application of CFG transformation, and show an overview of the CFG transformations for this technique in Figures 6.1 and 6.2. We begin with the original CFG, represented as CFG, and the user-modified CFG, CFG'; from CFG' we derive a set T of transformations that the user applied.

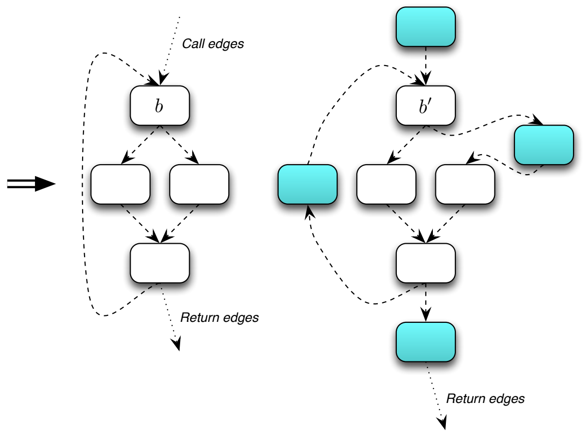
Next, we identify a region $R \subseteq CFG$ of blocks to copy, as shown in Figure 6.1a. At a minimum, R must contain all blocks whose out-edges were redirected or whose sizes would otherwise have increased; however, R may contain other blocks as well. We discuss the tradeoffs involved in selecting these additional blocks below.

We then copy R and apply the set T of CFG transformations to the copy, creating a modified region R'; this is shown in Figure 6.1b. We use the code relocation technique described in the next chapter to allow us to move R' without changing the behavior of the moved code. This step leaves the original region R unmodified; instead, we will logically replace R with R'.

Finally, we replace selected blocks in R with interception branches to blocks in R', as shown in Figure 6.2. This step requires three things. First, a mechanism to select the set $I \subseteq R$ of blocks that will be overwritten with interception branches. Second, a way to map between blocks in I and blocks in R'. Third, a mechanism to represent this code patching in the CFG. We select a block $r \in R$ to be be in I if it satisfies one of the following criteria. First, it has an



(a) Copying CFG to create CFG'



(b) Applying transformations to CFG'

Figure 6.1: Example of region-based code replacement. Instead of modifying the code corresponding to the original CFG in place, we instead copy the original CFG (Figure a) and modify the copy (Figure b). This figure continues in Figure 6.2.

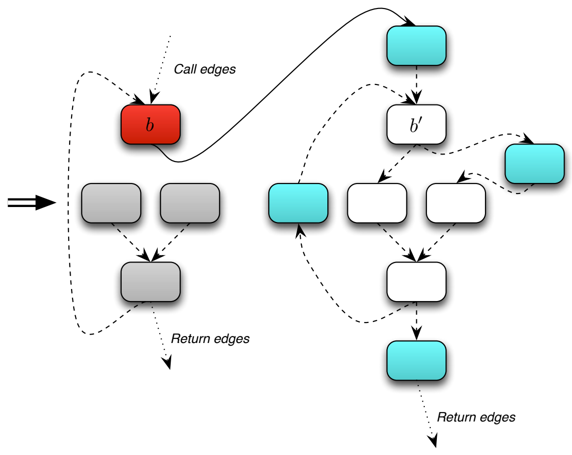
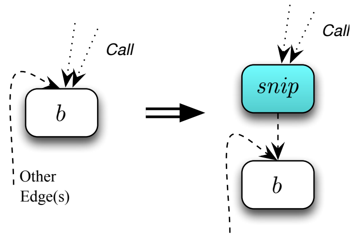
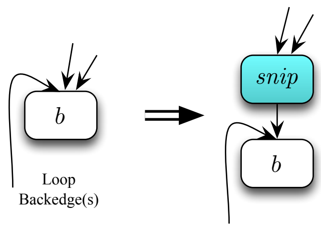


Figure 6.2: Example of region-based code replacement, continued. After copying the original CFG (Figure 6.1), we use an interception transformation to replace the entry block b of the original CFG with a branch to the corresponding location in the modified CFG. We shade b in red to indicate that it has been replaced by a branch, and shade the remaining original blocks in gray to indicate that they will no longer be executed.



(a) Function Entry



(b) Loop Entry

Figure 6.3: For reference, our transformations for inserting code at function entry and loop entry. In these two cases, we map the original block to the inserted snippet block rather than to the copy of the original block.

in-edge from a block outside of R; that is, if r is an entry block of R. Second, if r corresponds with a return address on the stack; such blocks are also logically entry points into R from other currently executing code. Third, if r is the entry block of a function, since we make the safe assumption that all functions may be reached by indirect call edges not represented in the CFG.

If we are modifying a running program, we use the debugger interface to write interception branches, as well as the new region R', to the binary. We assume this interface ensures that these changes take immediate effect by flushing the instruction cache; this is true of every modern operating system we have evaluated, including Linux, AIX, Solaris, FreeBSD, and Windows. If this is not the case, or if these techniques are applied from within the executing program, an explicit flush of the cache may be necessary. In addition, we assume this interface allows us to make arbitrary writes to the program. If certain pages are not writeable, such as to prevent code rewrite attacks, we first disable write protection, write R' and the set of interception branches, and then re-enable write protection.

We perform the mapping by extending our graph transformations to identify the equivalent blocks. If a transformation does not add a block this is a straightforward process. This is not the case for three transformations: function entry code insertion, loop entry code insertion, and function unsharing. For the two code insertion transformations, as shown in Figure 6.3, we branch from the original block to the inserted snippet block; this ensures that the snippet block will be executed. For function unsharing, we branch from each function's entry block to the appropriate copy.

We perform the code patching with another graph transformation, *interception transformation*, which replaces the contents of a basic block with a branch to a specified destination block; we show this transformation in Figure 6.4. This transfer models the effects of patching original code with a interception branch. We apply this transformation to the blocks in I, targeting their equivalent blocks in R'.

Once we have created the modified R' and identified the blocks in I that will be patched, we generate corresponding binary code. In general, standard code generation techniques, such as updating the target address of a branch,

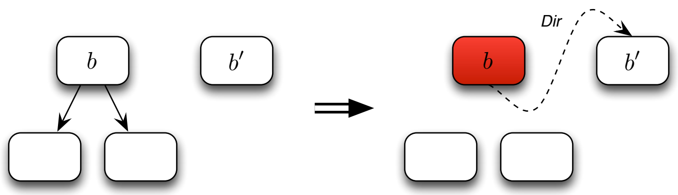


Figure 6.4: Our interception transformation, which replaces the contents of a given block, here represented b, with a branch to a specified destination block, here represented b'. In essence, this transformation redirects all in-edges to b to target b' instead. We shade b in red to indicate that its contents are overwritten with an interception branch.

suffice to generate code corresponding to R'. In some cases we may need to add additional branch instructions, such as if the target of a fallthrough edge is no longer laid out immediately after the source of the edge; in these cases we use techniques derived from compile-time modification toolkits [29]. We use the code relocation technique described in the next chapter to allow us to move this generated code to the end of the binary instead of its original location in the binary. Finally, we overwrite each block in I with branches.

As mentioned above, R must contain any block whose out-edges were redirected or whose size would be increased. However, we frequently include additional blocks in R. We do this for two reasons. First, if R is small, we may not be able to fit the branches to R'. For example, on x86-64 or PowerPC a single branch instruction cannot reach anywhere in the address space. Instead, we must first calculate the destination address in a register or on the stack and branch using that address; we show examples of such code in Figure 6.5. Such sequences may not fit in a single block. If we increase the size of R we can use the space occupied by the additional adjacent blocks to fit such branch sequences.

The second reason is efficiency. Executing an interception branch imposes surprising overhead, as we discuss in Appendix A. By including additional blocks in R (and thus R') we may be able to reduce how frequently interception branches are executed; this, in turn, can reduce overhead. In our experience, selecting code on a function basis results in lower overhead than selecting

```
1000: jmp $332200fb
```

(a) IA-32 Branch Sequence

```
1000: push $7766
1004: push $5544
1008: push $3322
100c: push $1100
1010: ret
```

(b) x86-64 Branch Sequence

```
1000: addis r0,r0,3322
1004: ori   r0,r0,1100
1008: mtlr  r0
100c: blr
```

(c) PowerPC-32 Branch Sequence

```
1000: addis  r0,r0,7766
1004: ori    r0,r0,5544
1008: rldicr r0,r0,32,31
100c: oris   r0,r0,3322
1010: ori    r0,r0,1100
1014: mtlr   r0
1018: blr
```

(d) PowerPC-64 Branch Sequence

Figure 6.5: Examples of interception branch sequences that can reach anywhere in the address space. We use 0x33221100 as an example destination on 32-bit platforms, and 0x7766554433221100 as an example destination on 64-bit platforms. Figure (a) shows an example branch on IA-32; since a single branch can use a 4-byte offset, it is possible to reach anywhere in the address space. Branch offsets are calculated from the end of the jump, so this branch uses an offset of (0x33221100 - 0x1005) = 0x332200fb. Figure (b) shows an example on x86-64. Like IA-32, x86-64 only supports 4 bytes of offset for a direct branch. We perform a longer branch by building the target address on the stack and using a return instruction. We use 16-bit pushes since they are not sign-extended; a 32-bit push is sign-extended to 64 bits. An alternative would be to load the destination address in a free register and use an indirect branch. Figure (c) shows an example for 32-bit PowerPC. This platform can encode 26 bits of offset in a branch, which again is not sufficient to reach anywhere in the address space. Instead, we must build the destination address in a register (`r0`), move it to the link register (`lr`), and branch. We assume `r0`, or another GPR, is available for use; if not, a GPR must be saved before the branch and restored at the destination. Similarly, either `lr` (or `ctr`, another special-purpose register) must either be free or saved. Finally, Figure (d) shows an example for 64-bit PowerPC; since only the lower 32 bits of a register is directly modifiable, we include a shift (`rldicr`) instruction to set the high 32 bits.

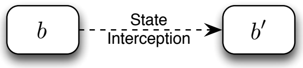
individual blocks within the function; we believe the improvement in overhead from only executing an interception branch at function entry dominates the increase in overhead from relocating additional code. However, we have not fully studied this relationship; for example, we may be able to further reduce overhead by moving multiple functions or a subregion of a function. We believe this to be an interesting area of further study.

6.2 State Interception

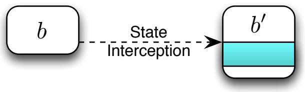
Region patching places interception branches at the entry points of modified regions of code. This technique is sufficient for binary rewriting or dynamic instrumentation of code that is executing outside such regions. However, it is not sufficient for replacing code that is actively executing; the process may continue to execute original code or execute part of an interception branch that overwrote multiple instructions. To handle this case, we augment region patching with another technique, *state interception*, that allows us to modify programs that are actively executing within a modified region by directly modifying process state to move execution from the replaced code to the replacement code. State interception is only necessary when code is first replaced; the interception branches inserted by region patching are used to intercept subsequent executions of the replaced code.

State interception only modifies the current program counter; all other aspects of program state, such as function pointers or return addresses on the stack, are handled by region patching by inserting additional interception branches, as discussed in the previous section. An alternative would be to update these values in memory; for example, we could rewrite the stack with the appropriate new return addresses. However, this approach requires significant analysis to be safe.

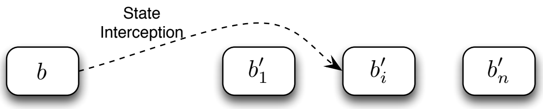
State interception has several similarities to region patching. Like region patching, we identify a mapping between the original code and its replacement. Instead of identifying entries to this region, though, we map from the current execution point (or points for multithreaded programs) to the corresponding point in the copied and modified code; this may be at an arbitrary instruction



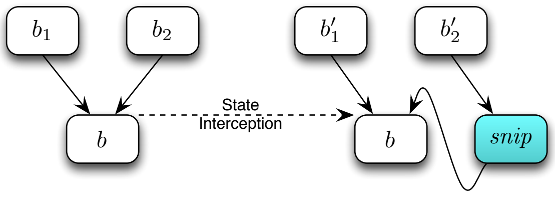
(a) Case 1



(b) Case 2



(c) Case 3



(d) Case 4

Figure 6.6: Examples of the four cases of state interception. We denote the interception operation with a dashed line from the location in original code to the location in modified code. Case 1 shows the straightforward case where we move from the original instruction i in a block b to corresponding instruction i' in the modified block b'. Case 2 shows our technique for handling inserted code; instead of moving to i' we move execution to the beginning of the inserted code. Case 3 shows our technique for handling cloned blocks; we select block b'_i (assuming an executing function f_i) as the destination. Case 4 shows our technique for handling code inserted along some, but not all, in-edges; we skip this code (unlike case 2) since we cannot guarantee it should execute.

rather than known entry points to the region. We begin by identifying the currently executing instruction i, and use this to identify the currently executing basic block b. We then map b to its equivalent modified block b', and use this to find the instruction i' in b' that is the equivalent of i.

We then consider the following cases:

1. b was not modified at the instruction i: transfer execution to i' in b'. This is the simplest case. We show an example of this case in Figure 6.6a.

2. b was modified to insert code immediately before i, such as for pre-instruction instrumentation: transfer execution to this code rather than i', since the code was specified to execute immediately before i. We show an example of this case in Figure 6.6b.

3. b was cloned as part of function unsharing. In this case, there will not be a single b'; instead, there will be a set $B' = \{b'_1, \ldots, b'_n\}$ with one block for each function that shared b. We examine the call stack to identify the currently executing function f_i, and use this function information to distinguish the appropriate $b'_i \in B'$ to use. We show an example of this case in Figure 6.6c.

4. New code was inserted before b and i is the first instruction in b. If the new code would be executed for all in-edges, then we transfer to the new code, as in the second case. If it would only be executed for some in-edges, such as code inserted at function or loop entry, we transfer to the entry of b' since we cannot guarantee that the new code should execute. We show an example of this case in Figure 6.6d.

Once we have determined the appropriate destination, we modify the program counter to the appropriate value.

This technique assumes that b' exists in the CFG. This may not be the case if the user modified the CFG to remove b. In these cases, we take the simple approach of skipping state interception; as a result, the program will continue to execute original code until it reaches an interception branch.

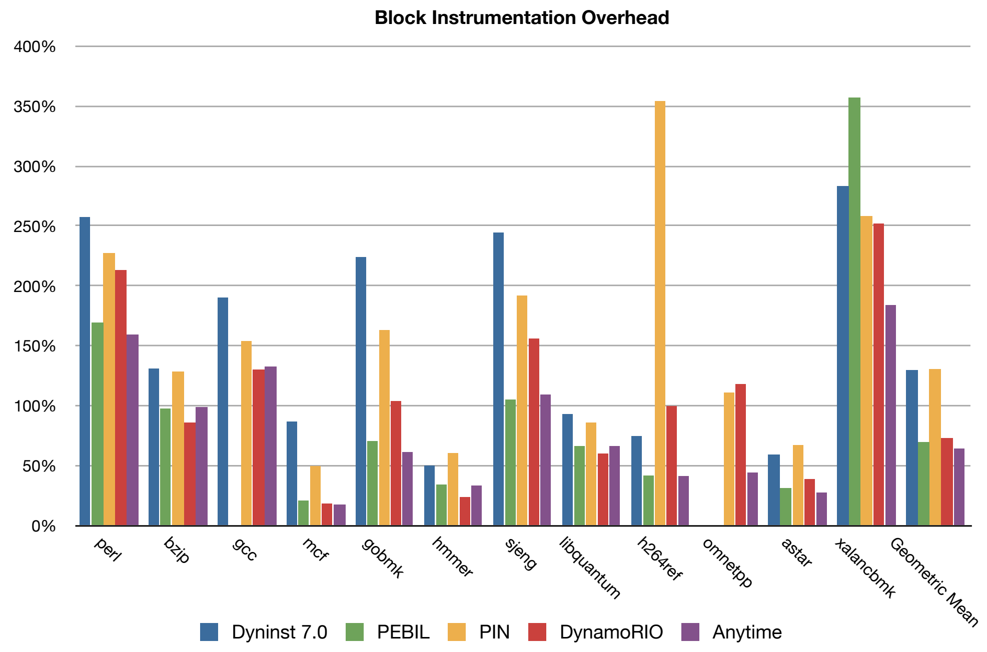


Figure 6.7: Performance of our techniques, Dyninst 7.0, PEBIL, DynamoRIO, and PIN. We used each toolkit to insert code into each basic block that increments a counter variable. The y-axis is the overhead of the instrumented binary. Missing values indicate the tool did not successfully instrument that benchmark; PEBIL failed on `gcc` and `omnetpp`, and Dyninst 7.0 failed on `omnetpp`.

6.3 Experimental Results

Region patching and state interception provide anytime code replacement while imposing proportional cost. We verified these characteristics with the following experiments. We began with an end-to-end experiment that shows the overhead involved in instrumenting each basic block. This experiment measures the total overhead imposed by this benchmark, which includes code replacement, relocation, and executing instrumentation. We selected this benchmark, rather than one that attempted to specifically measure the cost of code replacement, for two reasons. First, instrumenting every basic block is a common benchmark used to compare binary modification toolkits. Second, it is impossible to separate the overhead imposed by code replacement from relocation in a real program, as code replacement without relocation will cause the program's behavior to differ and thus invalidate any measurements we might take.

Next, we demonstrate that our approach imposes proportional cost by instrumenting a subset of basic blocks and measuring the overhead resulting from executing a partially instrumented program. We compare this overhead to other patch-based tools, which display similar proportional cost characteristics, and JIT-based tools, which impose fixed overhead even when no instrumentation is inserted.

Our final experiment demonstrates that we can instrument actively executing code. We do this by stopping an executing program and inserting instrumentation code immediately after the currently executing instruction. Our state interception technique ensures such instrumentation is executed immediately, while other approaches either delay execution until the next time the replaced region is entered or fail to execute the instrumentation entirely.

For our first experiment, we instrumented each program in the SPEC2006 integer suite to count how many basic blocks were executed. We compared our approach with Dyninst 7.0 [11], PIN [41], DynamoRIO [10], and PEBIL [38]. We did not compare our overhead to that of the popular Valgrind tool [50], as previous research has shown that its overhead is higher than both PIN and DynamoRIO. Similarly, we did not include SecondWrite [52], as it is not publicly available. We measured the total time to instrument and execute each benchmark.

The performance results for this experiment are shown in Figure 6.7; the y-axis is the overhead imposed. The overhead incurred by anytime code replacement is competitive or better on all benchmarks, with the exception of `bzip` where DynamoRIO incurred lower overhead; we believe this is due to the improved code layout algorithms DynamoRIO uses. Dyninst 7.0 performs worse due to its more frequent execution of interception branches, which particularly harmed it on the xalancbmk benchmark due to its heavy use of small functions, and failed on the `omnetpp` benchmark since `omnetpp` relies on exceptions, a language feature Dyninst 7.0 does not handle correctly. While Dyninst represents exceptions in the CFG as additional edges, its code reloca-

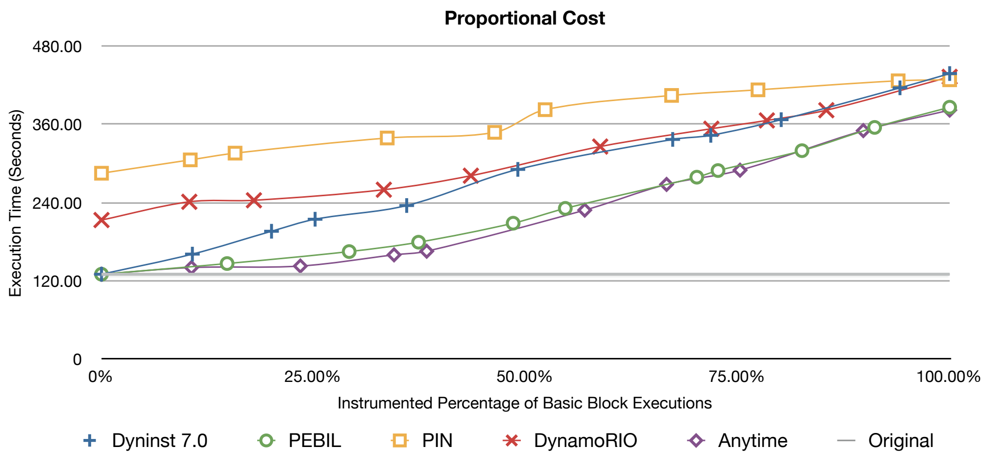


Figure 6.8: Performance of each toolkit on partially instrumented programs. The x-axis shows the percentage of total block executions that were of instrumented blocks, and the y-axis is overhead normalized to the overhead of full instrumentation for each toolkit.

tion mechanism causes exception handling to fail because it modifies the return addresses this mechanism relies on. We address this problem in Chapter 7.

The PEBIL rewriter incurs overhead close to our techniques on many benchmarks; this is unsurprising since the design of PEBIL is derived from Dyninst 7.0. It performs substantially worse on the `xalancbmk` benchmark, produced an invalid `gcc` program that crashed when executed due to data corruption, and failed to correctly instrument `omnetpp` for the same reasons as Dyninst 7.0. The DynamoRIO overheads are competitive due to their focus on instrumentation efficiency. Finally, PIN performs worse than other toolkits with the exception of Dyninst 7.0. We believe this is due to their more conservative code relocation mechanism; this was particularly apparent on the `h264ref` benchmark, which executes a large number of calls and returns that PIN emulates instead of executing natively.

For our second experiment, we determined the overhead of partially instrumenting the program. We did this by instrumenting randomly chosen blocks in the `perl` benchmark with a simple counter. We normalized the overhead of each execution by the ratio of instrumented block executions to uninstrumented block executions; the data is graphed in Figure 6.8. The y-axis shows the total time for each run, with an unmodified running time of 130 seconds. The x-axis is the percentage of total basic block executions that were of instrumented blocks. The overhead imposed by each binary modification toolkit decreases as fewer instrumented blocks are executed. The overhead imposed by anytime code replacement, Dyninst 7.0 [11], and PEBIL [38] decreases to zero, with our technique providing slightly lower overhead. The overhead imposed by DynamoRIO [10] and PIN [41] decreases as well, but to a fixed cost imposed by their JIT-based code replacement technique.

For our third experiment, we verified the timeliness of each toolkit by determining whether instrumentation inserted immediately after the current instruction would execute immediately. We did so as follows. For anytime code replacement and Dyninst 7.0 [11], we constructed a sample program consisting of a loop that increments a variable and then performs some other work. Our instrumenter waits for a random amount of time, stops the sample program, reads the value of the loop variable directly, inserts instrumentation at the current instruction to also read the value of the loop variable, and continues the sample program. When executed, anytime code replacement reports identical values for the loop variables; thus, the inserted instrumentation executed immediately. Dyninst 7.0 reports different values, with the instrumentation reporting a value one larger than the directly read value; thus, the instrumentation was executed in the next loop iteration. We did not perform this experiment on PEBIL, as it does not provide dynamic instrumentation.

PIN does not provide the ability to insert instrumentation at an arbitrary time; instead, instrumentation must be inserted before the code has executed the first time. We approximate the behavior described above with their block invalidation mechanism, which triggers re-instrumentation of a sequence of code. This invalidation is triggered by a second sequence of instrumentation which mimics the stop behavior described above. When executed, the loop was never re-instrumented despite our request. We believe this is a side-effect of the PIN code cache behavior; PIN only checks for invalidations lazily, such as when new code is parsed and instrumented. This does not occur in a loop. We did not perform this experiment on DynamoRIO, as they do not provide a re-instrumentation capability.

6.4 Summary

We presented a patch-based code replacement technique that provides anytime code replacement and proportional cost. Patch-based code replacement can be used at any point in the execution continuum without imposing the non-proportional overhead imposed by JIT-based code replacement or requiring the extensive semantic information required by in-place code replacement. We extended previous work with two new techniques: region patching, which lowers the overhead of code replacement, and state interception, which allows us to replace code that is being actively executed.

When compared with existing techniques, anytime code replacement provides the novel ability to modify actively executing code while imposing similar or lower overhead to other methods and preserving the proportional cost expected of patch-based instrumentation. In the next chapter, we discuss our sensitivity-resistant code replacement technique that allows us to perform patch-based code replacement without affecting the behavior of the original code.

7 Sensitivity-Resistant Code Relocation

In the previous chapters, we presented two techniques. The first, structured binary editing, used graph transformations as an abstraction for binary modification. The second, anytime code replacement, allows the incorporation of new or modified code into the binary. In this chapter, we discuss *sensitivity-resistant code relocation*, a technique for transforming original program code to compensate for the effects of instrumentation or code replacement on that code. Code replacement adds new code to the binary, moves original code, and may overwrite original code with branches to the new code. These operations change the contents of registers and memory as well. For example, moved code perceives a different value of the program counter, and overwriting code also changes the contents of memory. These effects, in turn, may alter the behavior of *sensitive* code that references changed registers and memory, causing the program to execute incorrectly.

Binary modification toolkits seek to compensate for the altered behavior of sensitive instructions using *code relocation*, a technique that produces code that has compatible visible behavior with the original code. Previous approaches have relied on ad-hoc definitions of sensitivity and visible behavior and thus impose unnecessary overhead [10, 41, 50] or may fail to preserve compatible behavior [10, 11, 36, 38, 41, 50, 52, 57, 68, 69]. We describe *sensitivity-resistant code relocation*, a technique that relies on a formal specification of sensitivity to both preserve correct behavior while often imposing lower overhead than previous approaches. This technique consists of four components: a model of instruction sensitivity, a formalization of the compatible visible program behavior that we wish to preserve, an analysis for identifying *externally* sensitive instructions that will alter this behavior, and an efficient compensation technique for handling such instructions.

This discussion assumes that the binary is only being instrumented, not modified. We assume such instrumentation does not explicitly modify the behavior of the original code; this assumption clearly does not hold for program modification. The techniques we describe in this chapter may be useful in certain cases of program modification, such as in forensic investigation of malware that prevents certain code sequences from executing [9]. Such changes take us into a semantic gray region and depend heavily on the expertise of the analyst to be performed correctly; however, they still have practical use. We believe an attempt to formalize the interaction of code relocation and explicit binary modification is an interesting area of future research.

We begin by introducing the notation we use in this chapter. In addition to the CFG, our analysis is performed over the data dependence graph (DDG). We also introduce a simple program model that treats a program as a function from input to output; we use this model to define the program behavior we wish to preserve.

Next, we introduce *compatible visible behavior*. Conceptually, we wish to ensure that code replacement does not change the externally visible behavior of the original binary, although its internal behavior may change and instrumentation may produce additional output. We formalize these characteristics as an extension of denotational semantics [65]. Two characteristics of our approach render the frequently-undecidable problem of determining semantic equivalence tractable. First, since code replacement does not delete code, there is a correspondence between each original basic block and a relocated basic block. Second, the execution order of the relocated code should be equivalent to the original code, since code replacement also does not alter this characteristic. We formalize these characteristics and define *output flow compatibility*, a stricter approximation of compatible visible behavior that uses these characteristics to simplify verification of compatibility.

We then define a model of instruction sensitivity. We define four classes of sensitivity to code replacement. Three of these categories represent instructions whose behavior is affected because their inputs are affected by modification: *program counter (PC)* sensitivity, *code as data (CAD)* sensitivity, and *allocated vs. unallocated (AVU)* sensitivity. The fourth category, *control flow (CF)* sensitivity, represents instructions whose control flow successors are moved. PC-sensitive instructions access the program counter and thus will perceive a different value when they are moved. CAD-sensitive instructions treat code as data, and thus will perceive different values if they access overwritten code, such as code replaced with an interception branch. AVU-sensitive instructions attempt to determine what memory is allocated by accessing unallocated memory and thus may be affected when new memory is allocated to hold moved and added code; we believe this is a newly identified sensitivity characteristic of programs. CF-sensitive instructions have had a control flow successor moved and thus may transfer control to an incorrect location. For each category of sensitivity we define how the inputs and outputs of the sensitive instruction are changed by modification. For example, a moved call is PC-sensitive, since it will save a different return address.

Adding code to a binary may alter its behavior in other ways, such as by introducing delays in execution, altering library or operating system state, or modifying the layout of the heap by allocating additional memory. We do not address these additional forms of sensitivity in this work; however, we believe that our model would suffice. For example, sensitivity to execution delays, such as timing checks, could be modeled by identifying the instructions that calculate the elapsed time and modeling the difference in the behavior of such instructions. Malware writers typically avoid such checks because they are difficult to implement and may be error prone. These difficulties stem from the complexities of processor execution models, and unpredictable memory delays and context switches.

Next, we define an analysis for identifying externally sensitive instructions. First, we define the characteristics of an instruction that make an instruction sensitive to code replacement. Second, we define a slicing-based analysis that identifies the instructions whose behavior may be affected by the different behavior of a sensitive instruction. Third, we define a technique based on symbolic evaluation that determines if this change in behavior will alter the overall program behavior. If it does, we conclude the instruction is externally sensitive.

We then define a technique for low-overhead preservation of original program behavior. We do this with the standard approach, by applying *compensatory transformations* to externally visible instructions. These transformations replace such instructions with sequences of code that emulate the original behavior of the instruction. These transformations are typically applied to each instruction individually [10, 36, 38, 41, 50, 52, 57, 68, 69]. This is not always the most effective approach; in some cases, it is possible to transform groups of instructions to further reduce overhead. We introduce an example, based on a technique from Dyninst 7.0 [11], that further reduces overhead when handling a code sequence often seen in position-independent code.

We conclude the chapter with a discussion of the results of this approach. First, we demonstrate that our approach is capable of handling defensive programs that explicitly attempt to detect any modification, and thus are extremely sensitive to code replacement. We do this by processing a simple test program with a set of *malware packers* and then instrumenting the result. Second, we compare the overhead imposed by our approach to the overheads imposed by other code relocation approaches.

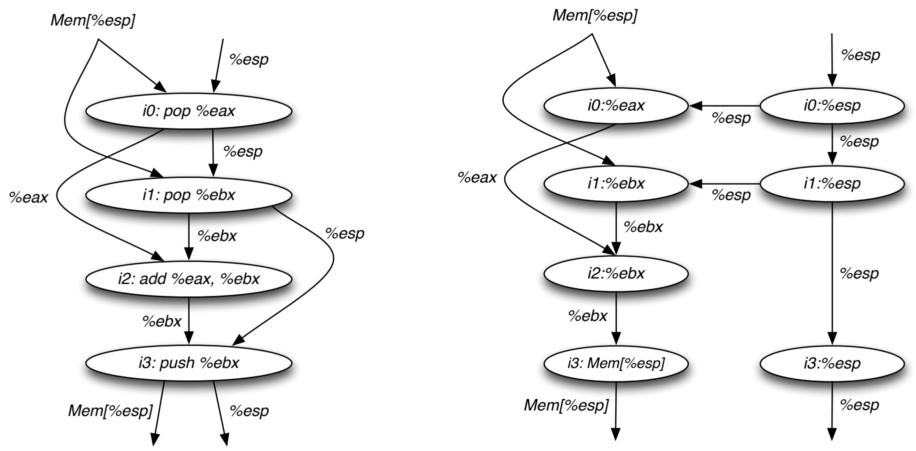
This chapter presents work that was done jointly with Kevin Roundy. In this work, we describe our contributions to this effort: the definition of compatible visible behavior and output flow sensitivity, as described in Section 7.2; the definition of instruction sensitivity, as described in Section 7.3, the external sensitivity analysis, as described in Section 7.4; and the implementation necessary for the experimental results on non-malware code. Kevin Roundy extended these techniques to defeat the sensitivity-resistance techniques used by malware programs. This extension involved developing efficient ways to compensate for CAD and AVU sensitivity, significant effort in implementing compensatory transformations for these binaries, and integration of these techniques into the SD-Dyninst research prototype [60]. Roundy's work is described in his dissertation [61].

7.1 Notation

We represent a binary program in terms of a process state, control flow graph (CFG), and data dependence graph (DDG). We use the CFG as defined in Section 3.1. We extend the conventional definition of a process state to include input and output spaces; this extension allows us to represent an input operation as a read from an abstract *input location* and an output operation as a write to an abstract *output location*. Finally, the conventional definition of a DDG over binaries [33] may over-approximate data dependences between instructions that define multiple locations, as is common on real machines (Figure 7.1a). We provide more precise dependence information by splitting such instructions into sets of single-definition *operations* and using these operations as nodes in the DDG (Figure 7.1b).

Process states (or simply *states*) are represented in terms of a set of abstract locations $\mathit{AbsLoc} = \mathit{Reg} \cup \mathit{Mem} \cup \mathit{In} \cup \mathit{Out}$. The registers Reg and memory Mem are defined conventionally; we assume the machine has a dedicated register pc that represents the program counter. For purposes of this work, we refer to the IA-32 architecture; however, we believe our approach is equally applicable to other architectures. We represent input and output with two sets of abstract locations, In and Out. The set $\mathit{In} = \{in_0, \ldots, in_m\}$ represents input to the program; we model each execution of an input operation (e.g., `scanf`) as an access of a unique input location. Output is similarly represented as the set $\mathit{Out} = \{out_0, \ldots, out_n\}$. A process state is a mapping from abstract locations to values; we use $\perp$ to represent an unallocated abstract location.

The input and output spaces of a program P are denoted $\mathit{In}_P \subseteq \mathit{In}$ and $\mathit{Out}_P \subseteq \mathit{Out}$, respectively. We define the function $\mathit{Execute}$ to relate program inputs and outputs as follows. Let the map $x : \mathit{In}_P \to \mathit{Values}_\perp$ represent an assignment of values to all locations in In_P; we refer to the set of all possible input assignments as Inputs_P. Then the output produced by executing P on x is denoted by $y = \mathit{Execute}(P, x)$ where the map $y : \mathit{Out}_P \to \mathit{Values}_\perp$ represents an assignment of values to all locations in Out_P. Executing a program traverses a path of nodes through the CFG; we denote this by $\mathit{ExecPath}(P, x)$, where $x \in \mathit{Inputs}_P$.



(a) Instruction Nodes (b) Operation Nodes

Figure 7.1: Data dependency graphs. Figure (a) illustrates the problems of representing instructions as single nodes. In this graph it is possible for paths to "cross" definitions; for example, there is a path from the definition of `%eax` by i_0 to the definition of the stack pointer `%esp` by i_3, when in the actual program there is no such dependence. Our extended DDG, shown in (b), makes the intra-instruction data dependencies explicit and excludes erroneous paths. For clarity, we omit the flags register `%eflags` and the program counter `%eip`.

We represent the data flow of a program with a data dependence graph (DDG). To provide more precise dependence information, we split instructions into a set of single-definition *operations*. These operations form the nodes of our extended DDG. Formally, the DDG of a program P is a digraph $DDG = (V, E)$. We use a virtual operation called *initial* to represent the initial assignment of abstract locations. A vertex in this graph is a pair of an instruction and an abstract location defined by that instruction, and the set of vertices is $V = \{(initial, a) | a \in AbsLoc\} \cup \{(i, a) | i \in P \wedge a \in defs(i)\}$, where $defs(i)$ represents the set of abstract locations defined by i. The set of edges $E \subseteq V \times V$ represents use-def chains between operations. We show an example of our extended DDG in Figure 7.1b. We represent the DDG of a particular program P as DDG_P.

7.2 Output Flow Compatibility

We formalize our intuition of visible behavior in terms of denotational semantics. Two programs have the same denotational semantics if, for the same input, they produce the same output [65]. Requiring strict semantic equivalence would not allow instrumentation to consume input or produce output; we address this limitation by assuming instrumentation code has its own input and output spaces and defining *compatible visible behavior* as denotational semantic equivalence over the input and output spaces of only the original program. We assume that the original and instrumented programs must have equivalent control flow in addition to compatible visible behavior. We formalize this in terms of a stricter approximation of visible behavior called *output flow* compatibility.

We define the original program P and instrumented P' to be visibly compatible, written $P' \sqsupseteq P$, if the following three conditions hold. First, $In_{P'} \supseteq In_P$, and all input locations in $In_{P'} \setminus In_P$ are only read by instrumentation code. Second, $Out_{P'} \supseteq Out_P$, and all output locations in $Out_{P'} \setminus Out_P$ are only written by instrumentation code. Third, for *compatible* inputs, P and P' produce *compatible* outputs. We define input compatibility as follows. Let $x \in Inputs_P$; then $x' \in Inputs_{P'}$ is compatible with x, written $x' \sqsupseteq x$, if $\forall l \in In_P, x(l) = x'(l)$. We define output compatibility in a similar way. Let $x \in Inputs_P$ be some input to P, and x' be an input to P' such that $x' \sqsupseteq x$. P and P' are output compatible if $\forall x, x', Execute(P', x') \sqsupseteq Execute(P, x)$.

We define output flow compatibility as follows:

Control flow constraint: The instrumented and original programs must, when executed on compatible inputs, traverse equivalent paths through the CFG (disregarding instrumentation). For simplicity, we assume that instrumentation is only inserted on a basic block boundary; we can always split blocks to ensure this is the case. Since instrumenting a program does not delete original code, there is a natural correspondence between each original basic block b_i and a basic block b'_i in the instrumented program. We do not consider the execution of instrumentation in our definition of control flow equivalence; we represent this with a function *Filt* that removes all blocks representing instrumentation from a path p' through $CFG_{P'}$. P and P' have equivalent control flow if $\forall x \in \mathit{Inputs}_P$ and $\forall x' \in \mathit{Inputs}_{P'} : x' \sqsupseteq x, \mathit{Filt}(\mathit{ExecPath}(P', x')) = \mathit{ExecPath}(P, x)$. This definition does not handle blocks that were split to insert instrumentation; for simplicity, we assume such blocks were also split in the original CFG.

Output constraint: The control flow constraint means P' and P will write to all output locations in Out_P in the same order. In addition, they must both write the same values. P' satisfies this constraint if the following holds for all inputs x to P and compatible inputs x' to P'. Let $\langle b_0, \ldots, b_m \rangle = \mathit{ExecPath}(P, x)$ and $\langle b'_0, \ldots, b'_n \rangle = \mathit{Filt}(\mathit{ExecPath}(P', x'))$; by the above constraint $m = n$. Then for each block pair $b_i, b'_i, 0 \leq i \leq n$, each output operation in b_i and b'_i must produce the same values.

7.3 Sensitivity Model

In this section, we present our model of operation sensitivity. We use operations rather than instructions to increase precision; it is possible for some operations within an instruction to be sensitive without all operations necessarily being sensitive. An operation is sensitive to code replacement if its behavior will be directly affected by replacement. This occurs in two ways. First, an input value to the operation may be changed, such as a moved operation that reads the program counter or an operation that accesses modified memory. Second, the required output of the operation may change, such as a branch whose successor was moved. We define four classes of sensitivity to the effects of code replacement: PC-sensitivity to code being moved, AVU sensitivity to code being added, CAD sensitivity to code being overwritten, or CF-sensitivity to a successor being moved.

We determine whether an operation is sensitive to code replacement using the algorithm shown in Figure 7.2. We consider two general types of sensitivity: the operation's (i.e., its containing instruction's) sensitivity to being moved and the sensitivity of the remainder of the program to being modified. Moving an operation changes its address but does not modify the operation in any other way. All inputs to the operation will be unaffected with the exception

```
Algorithm: IsSensitive
input  : An operation $o$ in an instruction $i$, a set $A$ of added memory, a set $M$ of
         moved instructions, and a set $O$ of overwritten memory
output : Whether $o$ is sensitive
if $i \in M \wedge$ Uses($o, pc$) then
    return true                                        // $o$ is PC-sensitive
if Uses($o, A$) then
    return true                                        // $o$ is AVU-sensitive
if Uses($o, O$) then
    return true                                        // $o$ is CAD-sensitive
if Defines($o, pc$) then
    foreach instruction $i' \in$ Successors($i$) do
        if $i' \in M$ then
            return true                                // $o$ is CF-sensitive
return false                                           // $o$ is not sensitive
```

Figure 7.2: An overview of our algorithm for identifying operation sensitivity to code replacement. We test for four classes of sensitivity: PC-sensitivity to being moved (line 2), AVU sensitivity to added code (line 4), CAD sensitivity to overwritten code (line 6), and CF-sensitivity to successors being moved (lines 8-10).

of the *pc*, which contains the address of the operation and thus will change. We define an operation to be *PC-sensitive* if it uses the program counter and its containing operation will be moved (line 2).

Allocating new memory or overwriting existing memory changes the contents of the corresponding abstract locations. This will affect all operations that use these abstract locations. We define an operation to be *AVU-sensitive* if it uses an abstract location that represents added memory (line 4); similarly, an operation is *CAD-sensitive* if it uses an abstract location that represents overwritten memory (line 6).

The final type of sensitivity is sensitivity to the program being modified. Moving an instruction also affects its immediate control flow predecessors. We also define this sensitivity in terms of operations, in this case, the operation that writes *pc*. We define an operation to be *CF-sensitive* if it writes *pc* and one or more of its successors will be moved (lines 8-10), since executing the CF-sensitive operation may cause the control flow of the instrumented program to diverge from that of the original.

It is straightforward to precisely identify both PC-sensitive and CF-sensitive instructions, as no pointer analysis is required. Identifying CAD and AVU-sensitive instructions requires pointer analysis to identify whether an operation reads from overwritten or added memory, respectively. However, it is safe to over-approximate an operation as sensitive, and thus our analysis is conservative.

7.4 External Sensitivity Analysis

In this section, we present an analysis for identifying whether an instruction is externally sensitive to the program being instrumented. As in the previous section, we describe this algorithm in terms of operations; an instruction is externally sensitive if it contains an externally sensitive operation. While it is straightforward to identify sensitive operations, not all sensitive operation will cause a change in overall program behavior. Identifying the *externally* sensitive operations that will do so requires analysis of the effects of a sensitive operations on the remainder of the program. For example, if we move a call as part of a region of code, the return address saved by the call will change; however, this changed address will correspond with the new location of the call and thus the control flow of the program will not be affected. However, if the return address is used for a different purpose, such as part of a pointer to data, changing the value may affect the program's behavior.

For example, consider the effects of changing the location of a function that contains a call instruction. Call instructions contain two operations: one that sets the *pc* and one that saves the return address. This return address will change if the call is moved, and thus the second operation will be sensitive to movement. If this return address is used only for returning from the call then the overall control flow of the program will not be altered, rendering the operation internally sensitive. However, if the address is used as a data value, such as in a pointer calculation, then moving the call may affect visible behavior, rendering the call externally sensitive.

We determine whether an operation is externally sensitive with the algo-

```
1 Algorithm: IsExternallySensitive
  input   : An operation $o$
  output  : Whether $o$ is externally sensitive
2 if isSensitive(op)) then
3     Let $S$ = ForwardSlice($op$) ;
          // $S$ stops at visible operations foreach visible operation $v$ in $S$ do
4         Let $c$ = ChangedResult($v$) ;
5         if ChangesOutputFlow($c$) then
6             return true
7 return false
```

Figure 7.3: An overview of our algorithm for determining whether an operation is externally sensitive. We begin by determining if the operation is sensitive to code replacement (line 2). If so, we determine its affected operations with a forward slice (line 3) and determine the changed results of each affected operation (lines 4-6). If these changed results will change output flow (line 7), we conclude the input instruction is externally sensitive.

rithm in Figure 7.3. First, we identify whether an operation is sensitive and skip those that are not (line 2). For each sensitive operation, we determine its forward *slice* (line 3); this slice includes the set of operations whose behavior may be affected by the sensitive operation [33]. We then examine each operation in the slice to determine whether it can affect output flow equivalence, either by changing control flow or an output value (lines 4 and 5); we call these operations *visible operations*. For each visible operation, we determine how code replacement would change its results (line 6) and identify whether the change in result (if any) might break output flow equivalence (line 7); in this case we conclude the original instruction is externally sensitive. We describe each of the major component functions below.

`IsSensitive`: We determine whether an operation is sensitive to code replacement using the algorithm shown in Figure 7.2 and discussed in the previous section.

`ForwardSlice`: Our analysis operates over the set of operations whose execution may be affected by the sensitive operation. Any operation that is not affected by the sensitive operation will not have its behavior changed and thus can not change the program's visible behavior. We define the *affected set* of operations affected by a sensitive operation o as the forward slice from o, terminating at *visible* operations that may affect output flow. This includes any operation that sets the program counter or produces output. We assume that a compensatory transformation will ensure that the effects of code replacement will not propagate past these points. As a result of this termination, we do not include control dependence edges in the slice. We discuss why this approximation is valid at the end of this section.

`IsVisible:` An operation is visible if it can directly affect control flow or output. All other operations can only affect internal elements of data flow and thus will not directly cause output flow equivalence to fail. We identify visible operations as follows. An operation affects control flow if it writes to pc; we call such operations *CF-visible*. Similarly, an operation affects output if it writes to an abstract location in Out_P; we call such operations *output-visible*. These are the only operations whose changed behavior we must model to determine whether output flow equivalence is affected by code replacement.

`ChangedResult:` Our analysis models how code replacement would change the output of each visible operation using symbolic execution. We do this as follows. First, we calculate the chop [31] $chop(o_s, o_v)$ from the sensitive operation o_s to the visible operation o_v. Second, we use symbolic execution to derive a symbolic representation of the chop. We represent the result as follows:

Symbolic Representation: The symbolic representation of a chop $chop(o_s, o_v)$ is a function

$$Sym_{o_s,o_v}(x_0, \ldots, x_n)$$

where $(x_0, \ldots, x_n)$ represent inputs to operations in the chop; the result of this function is the value produced by o_v.

Third, we derive an expression of how code replacement would change the output of o_v as follows. We determine the *input difference* for each input x_i:

Input Difference: We represent how code replacement will change the value of x_i with a function f_i; these functions are defined below.

The new output of o_v will be $Sym_{o_s,o_v}(f_0(x_0),\ldots,f_n(x_n))$. Fourth, we determine if there exists a binding of values to inputs that will cause o_v to break output flow equivalence; if this is true then we conclude o_s is externally sensitive. We describe each of these steps below.

We derive the chop $chop(o_s, o_v)$ with a forward traversal of the DDG and use symbolic evaluation to derive Sym_{o_s,o_v} [16]. We determine how code replacement will change the inputs to the chop as follows. For each $x_i \in \{x_0, \ldots, x_n\}$ we define the mapping function f_i as follows. If code replacement will not change the value of x_i then f_i is the identity; this is the case with any input to an operation other than o_s. Otherwise, f_i depends on what class of sensitivity o_s belongs to:

PC-sensitive: If o_s is PC-sensitive then x_i represents the pc. Let i represent the moved instruction, a its original address, and a' its new address. Then x_i must equal a, and $f_i(x_i) = a'$.

AVU-sensitive: If o_s is AVU-sensitive then x_i represents an abstract location a in memory that was added by code replacement. In this case $x_i = \perp$, as this memory originally was unallocated, and $f_i(x_i) = v'$ where v' represents the new value written into a by code replacement.

CAD-sensitive: If o_s is CAD-sensitive then x_i represents an abstract location a in memory that was overwritten by code replacement. In this case $x_i = v$, where v represents the original contents of a; we assume that such memory is read-only and thus v is known. Instrumentation would overwrite a new value v' into a; thus $f_i(x_i) = v'$.

CF-sensitive: Unlike the previous three cases, the inputs to a CF-sensitive operation will not be changed by code replacement unless the operation also depends on a PC, AVU, or CAD sensitive operation. Thus $f_i(x_i) = x_i$ for all inputs.

`ChangesOutputFlow`: The final step in our external sensitivity analysis determines whether code replacement would cause a visible operation to change the program's control flow or output. Clearly, changing the value produced by an output-visible operation breaks output flow equivalence. However, this is not necessarily the case for CF-visible operations. The values written to pc by these control flow operations may change without changing the control flow of the program (and thus breaking output flow equivalence) so long as any changes precisely correspond with the movement of a control flow successor. Consider the call example from above. In this example, the return address stored by the call would be changed by code replacement since the call is moved; this change will cause the corresponding return instruction to write a different value to pc. However, since this new value is the new address of the call's successor, the control flow of the instrumented program would not be changed and thus the call is not externally sensitive.

We consider the following two cases:

Output-visible: As we mention above, any change in output will break output flow equivalence. Therefore, if there exists an assignment of values to $x_0, \ldots, x_n$ such that

$$Sym_{o_s,o_v}(f_0(x_0), \ldots, f_n(x_n)) \neq Sym_{o_s,o_v}(x_0, \ldots, x_n)$$

then o_v is *output flow breaking*, and thus we would conclude that o_s is externally sensitive.

CF-visible: This case is more complex as we must account for the movement of instructions. Recall that an output-visible operation o_v sets pc, and therefore we can treat the value it produces as an address. Let $Move$ be a mapping function from the original address of an instruction to its moved address. For example, if an instruction i was moved from an address a to an address a' then $Move(a) = a'$; if i was not moved then $Move$ is the identity. Then o_v is output flow breaking if there exists an assignment of values to $x_0, \ldots, x_n$ such that

$$Sym_{o_s,o_v}(f_0(x_0), \ldots, f_n(x_n)) \neq Move(Sym_{o_s,o_v}(x_0, \ldots, x_n))$$

Our external sensitivity analysis relies on static slicing and symbolic evaluation; both of these techniques are notoriously imprecise and expensive when applied to binaries. We handle imprecision by being overly conservative, since it is always safe to falsely assume an operation is externally sensitive. We reduce the expense of slicing by sharply limiting the size of the slice, since we terminate slices at any visible operation. Since control flow instructions are by definition visible operations, this ensures the slice will contain no control-dependence edges. By eliminating control-dependence edges, we greatly reduce the cost of symbolic evaluation, as we do not have to consider the effects of multiple possible execution paths.

7.5 Efficient Compensation

The final step in our sensitivity-resistant code relocation algorithm is to transform the instrumented program to preserve its original visible behavior. We do this by applying a *compensatory transformation* to externally sensitive instructions affected by code replacement. This transformation must preserve the original visible behavior of the transformed code and avoid imposing unnecessary overhead. We describe three transformation strategies, *instruction transformation*, *group transformation*, and *control flow interception*. Instruction transformation replaces each externally sensitive instruction with code that emulates its original behavior; all other instructions are left unchanged. This strategy is derived from the ad-hoc transformations used by previous work [10, 11, 41, 42, 50], and is described in Section 9.

Group transformation preserves the behavior of a group of instructions

```
input  : A region $R$ of instructions
output : A new region $R'$ of transformed instructions
1 Let $R' = \emptyset$ ;
2 foreach externally sensitive instruction $i \in R$ do
3     if $\exists$ a group $G$ and group transformation $GT$ of $i$ then
4         $R' = R' \cup GT(G)$ ;
5         Mark instructions in $G$ as transformed
6     else
7         Let $T$ be an instruction compensation transformation ;
8         $R' = R' \cup T(i)$
9 return $R'$
```

Figure 7.4: An overview of our transformation algorithm. For each externally sensitive instruction, we first attempt to identify a group G that includes i (line 3). If such a group exists, we apply the appropriate group transformation GT and mark the entire group as transformed (lines 4-5). If not, we apply an instruction transformation (lines 6-8). Section 9 describes instruction transformations and Section 9 describes group transformations.

rather than each instruction individually. We describe an overview and proof of concept of this approach in Section 9. This strategy is a generalization of the approach used by Dyninst 7.0 [11], which recognizes and transforms pre-defined patterns of instructions. Our algorithm instead uses our symbolic representation of the code to determine a correct and efficient transformation. In addition to the lower overhead offered by transforming a group of instructions, we can further reduce overhead by applying code optimization techniques such as constant propagation (e.g., of *pc*), function inlining, or partial evaluation.

Control flow interception is used by anytime code replacement and other patch-based code replacement approaches [11, 36]. This control flow interception strategy handles CF sensitive instructions by overwriting the original locations of moved code with branches to their new locations. Thus, during execution, the instruction will transfer control to the original address of a moved instruction and the branch will redirect execution to its moved location. Since this strategy overwrites original code it may affect the behavior of CAD sensitive instructions. However, it results in significantly lower overhead than

Sensitivity	Original Instruction	Transformed Instruction
PC-sensitive	call `foo`	push $orig
		jmp `foo`
	push $offset(%rip)	push $(offset-delta)(%rip)
CAD-sensitive	mov (%eax), %ebx	cmp %eax, $textEnd
		jge L1
		mov $offset(%eax), %ebx
		jmp L2
		L1: mov (%eax), %ebx
		L2: . . .
CF-sensitive	jmp %eax	jmp %eax

Figure 7.5: Examples of instruction transformations for PC, CAD, and CF sensitive instructions. The PC transformation is derived from current binary modification toolkits [10, 11, 41]. The PC sensitive call instruction is transformed by splitting it into two operations that piecewise emulate the call. We transform PC sensitive memory accesses by subtracting the distance $delta$ the instruction was moved from the encoded offset. The CAD sensitive move instruction is transformed by redirecting memory accesses to overwritten code (bounded above by $textEnd) to a copy of the code by adding $offset. For simplicity, this example assumes no data resides at a lower address than modified code. The CF sensitive indirect jump is not transformed; we assume the instruction sequence that generates a value in `%eax` is emulated. When executed the jump will transfer back to original code, where an interception branch will capture execution.

an instruction transformation of the CF sensitive instruction.

Our compensatory transformation algorithm is shown in Figure 7.4. We iterate over each externally sensitive instruction in the region. We first determine whether it has a known group transformation; if so we apply the appropriate transformation (lines 3 through 5) and mark all other instructions in the group as transformed (line 5). Otherwise, we apply the appropriate instruction transformation (lines 7 through 8). We do not show the use of the control flow interception strategy to handle CF sensitive instructions since it may modify code outside the provided region.

These transformations may further modify the program. For example, these transformations frequently increase the size of the input code and thus

Original Code	Instruction Transformation	Group Transformation
main:		
i_1: call `thunk`	i_{1a}:**push $(orig)**	i_1: call `thunk`
	i_{1b}:**jmp `thunk`**	
i_2: add $(tOff), %ebx	i_2: add $(tOff), %ebx	i_2: **add $(tOff - delta), %ebx**
thunk:		
i_3: mov (%esp), %ebx	i_3: mov (%esp), %ebx	i_3: mov (%esp), %ebx
i_4: ret	i_4: ret	i_4: ret

Figure 7.6: Example of a thunk group transformation applied to an IA-32 jump table fragment. Transformed code is shown in bold. The original code calculates a pointer by using `thunk` to access the current PC and adding an offset. Instruction transformation will emulate the call i_1 as shown in Figure 7.5; $orig$ represents the original return value. While i_4 is not transformed under this approach, we will require an interception branch at the original address of i_4. Group transformation results in only i_2 being transformed; $delta$ represents the distance i_1 was moved, and no interception branch is required at i_4.

may require that the transformed code be moved. This movement, in turn, may increase the number of sensitive instructions. We handle this problem by iterating until the set of moved code converges.

Instruction Transformations

Instruction transformation replaces each externally sensitive instruction i with a new sequence that emulates its original behavior; all other instructions are left untransformed. We implement this strategy with a translation table that maps from an input externally sensitive instruction to a replacement code sequence. Examples of such transformations for PC, CAD, and CF sensitive instructions are shown in Figure 7.5. We discuss the transformation of these and AVU sensitive instructions below.

PC sensitive instructions have the original and changed values of pc differ by a constant. If the value of pc is known when the code is transformed (e.g., at runtime) we simply replace the new value with the original (as shown).

Otherwise, we subtract the distance the instruction was moved to recover the original value.

We transform CAD sensitive instructions by making a copy of the modified regions of code; accesses to these addresses are redirected to the copy while accesses of other addresses are not modified.

CF sensitive instructions must be transformed to account for movement of their successors and not necessarily changes in their inputs. The distance each successor has been moved is frequently different due to the presence of inserted instrumentation. Therefore there is no linear function that can be used to convert from original to moved addresses. We transform the instruction to jump to its original address and add interception branches at those addresses [11]. An alternative approach is to use a hash table to perform this conversion [10, 41, 42, 50].

Transforming AVU instructions is more complex. Attempting to access unallocated memory will cause a fault. We need to emulate this fault instead of emulating the original output of the instruction, which we do by using a fault handler interposition approach similar to that of DIOTA [42]. We redirect the memory access to read from an illegal address (typically 0); this causes the operating system to report a fault to the process. However, the reported address will be incorrect. We address this by interposing our own fault handler. This replacement handler intercepts the fault, emulates the original fault information (e.g., faulting instruction address and accessed memory address), and calls the original fault handler.

Group Transformations

Group transformation preserves the overall behavior of a group of instructions rather than that of each instruction in the group. As we show in Figure 7.6, instruction transformation may unnecessarily emulate instructions, resulting in unnecessary overhead. This is due to that strategy's limited scope; it considers transforming only externally sensitive instructions. Group transformation addresses this problem by considering instructions that are not externally sensitive for transformation. In this work, we characterize group transformation

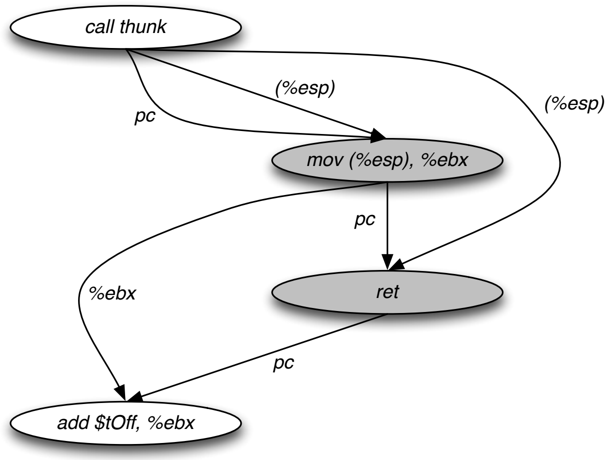


Figure 7.7: The group of instructions for the example in Figure 7.6. The shaded nodes are included for clarity but are not included in the group as they may be called from other locations as well. The externally sensitive call instruction has three outputs that must be preserved. Defining these instructions as a group reduces this to a single output.

and provide a motivating example; in future work we intend to implement and test this concept.

A group transformation algorithm has two requirements: selecting each group G of instructions to transform and generating the replacement group G'. Selecting groups rather than individual instructions is key to improving performance.

We first define a group. Intuitively, a group containing an externally sensitive instruction i consists of all instructions that can be modified to compensate for the changed behavior of i, but whose modification will not affect the instructions outside of the group. Such instructions can be transformed without causing unintended side-effects. We formalize this intuition in terms of the DDG. A set of operations O is an *op-group* of an externally sensitive operation s if all operations $t \in O$ are dominated by s (if s is data sensitive) or post-dominated by s (if s is control sensitive), and the corresponding instruction group G consists of all instructions that contain an operation in O. Consider the example code shown in Figure 7.6; for this code, the group consists of i_1

and i_2 as shown in Figure 7.7.

A group transformation algorithm must select a group G for each externally sensitive instruction i and construct a replacement group G' that has the same behavior as G. Selection is done as described above. Our proof of concept implementation both identifies G and constructs G' using a set of templates. Picking a beneficial group G is an open problems; if G is too small then we miss opportunities for reducing overhead, and if G is too large then constructing G' is made more difficult. Similarly, constructing G' from a general group G is an open problem. We believe it should be done using the DDG and can leverage compiler optimization techniques, but have not done any work in the area.

7.6 Experimental Results

Our code relocation algorithm properly preserves the semantics of the instrumented program while frequently reducing the overhead imposed by code replacement. We verified these characteristics with the following experiments. First, we instrumented several tamper-resistant malware programs to show that we properly compensate for attempts by a program to detect modification to the contents or shape of its address space. Second, we instrumented the SPECint 2006 benchmarks, Apache, and MySQL to show that our code relocation algorithm results in lower average overhead than either the Dyninst 7.0 or PIN binary modification toolkits. We chose Dyninst 7.0 since several of our techniques are derived from this toolkit, and PIN as an example of a conservative code relocation approach.

We implemented our algorithm in the Dyninst 7.0 binary modification toolkit, creating the SR-Dyninst research prototype. We identify sensitive instructions using information provided by the InstructionAPI component of Dyninst 7.0, and built a new symbolic evaluation and slicing component to assist in our identification of externally sensitive instructions. This component uses a semantic instruction model provided by the ROSE compiler suite [58]. While these experiments were done in the context of the Dyninst 7.0, the techniques and software we built can be used to extend other toolkits, such as PIN, to have the same capabilities as those we added to Dyninst 7.0.

Packer Tool	Market Share	CAD Sensitive	Anti-Debug	Success
UPX	9.45%			yes
PolyEnE_CAD	**6.21%**	**yes**		**yes**
EXECryptor	4.06%	yes	yes	
Themida	2.95%	yes	yes	
PECompact_CAD	**2.59%**	**yes**		**yes**
UPack	2.08%			yes
nPack	1.74%			yes
ASPack	1.29%			yes
FSG	1.26%			yes
Nspack	0.89%			yes
ASProtect	**0.43%**	**yes**		**yes**
Armadillo	0.37%	yes	yes	
Yoda's Protector	**0.33%**	**yes**	**yes**	**yes**
WinUPack	0.17%			yes
MEW	0.13%			yes

Figure 7.8: SR-Dyninst applied to the binary protection tools that are most prevalent in malware, with optional CAD features (e.g., self-checksumming, data masquerading as code) enabled for PolyEnE and PECompact. Packers successfully instrumented by SR-Dyninst are labeled in bold. Gaps in the table represent packers with anti-debugging techniques that are unrelated to sensitivity analysis and that we have yet to defeat.

Tamper-Resistant Binaries

To demonstrate that we can safely instrument CAD and AVU-sensitive programs, we incorporated the SD-Dyninst research prototype built by Kevin Roundy [60] into SR-Dyninst. Malware typically uses three categories of anti-analysis techniques: attempts to hinder static analysis (e.g., control flow obfuscation and runtime code modification), attempts to resist tampering with the binary code by adding CAD sensitivities (e.g., self checksumming), and attempts to detect the presence of an analysis tool (e.g., detecting the presence of a debugger). SD-Dyninst is focused on defeating packers that resist static analysis or use anti-debugging techniques and does not correctly instrument binaries that employ CAD sensitivity; SR-Dyninst overcomes this limitation.

We generated samples that exhibit the same defensive techniques used by malware by applying the packer tools that are most popular with malware authors to a sample program. We used these synthetic test programs rather than actual malware to allow us to test our work without the complexity of operating directly on malware. SD-Dyninst's previous evaluation used the default settings of the packer tools. To demonstrate that our approach would successfully handle CAD-sensitive binaries, we enabled all features in the packer tools that would add CAD sensitivity to the packed binaries. The results listed in Figure 7.8 show that SR-Dyninst successfully instrumented four of seven packed binaries that had defeated SD-Dyninst with sensitivity. We believe our failure to instrument the remaining three packers is due to their anti-debugging features rather than their CAD sensitivity, and are working to overcome these features.

Performance Results

We also measured the performance impact of our new relocation technique on the execution time of instrumented programs. Our scenario includes the cost of code replacement and relocation without inserting any additional code that would impose its own overhead. Unlike the results presented in Section 6.3, this is not an end-to-end study; instead, we wished to focus on only the overhead imposed by code relocation.

Our performance experiments were run on an input set of binaries consisting of the SPECint 2006 benchmark suite, Apache, and MySQL. Each of these programs was built from source with default settings. We instrumented both the program binary and any libraries on which it depended. We ran the SPECint suite using reference inputs and tested Apache and MySQL with their provided benchmarking tools. These experiments were run on a 2.27 GHz Intel quad-core Xeon machine with 6GB of memory.

We measured the execution overhead caused by executing moved and transformed code instead of original code. In our experiments, we forced relocation of every basic block in the binary. For Dyninst 7.0 and SR-Dyninst we measured execution time. For PIN, we measured dynamic translation and

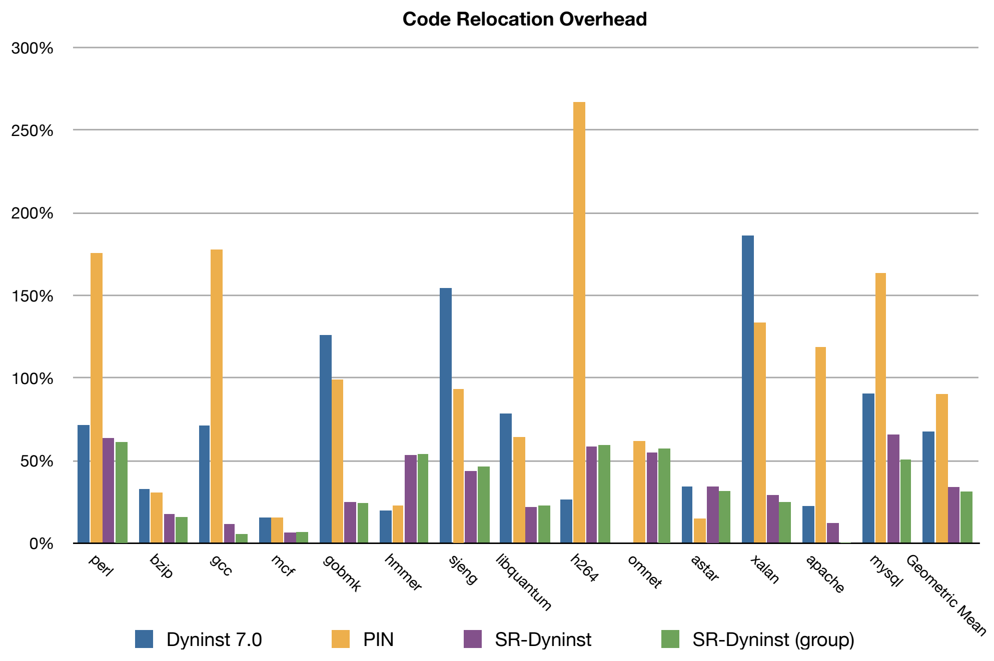


Figure 7.9: Performance of our approach compared to Dyninst 7.0 and PIN. We show two sets of results for our approach. The first uses only instruction transformations, while the second includes the thunk group transformation of Section 9. The y-axis is execution time normalized to the unmodified execution time.

execution time, since the toolkit does not provide any way of distinguishing these values. However, from their previously published results [41], the cost of dynamic translation is small for long-running benchmarks and thus we do not believe it significantly impacts our results.

The performance results are shown in Figure 7.9. The y-axis is the execution time normalized to the uninstrumented run time (0%). SR-Dyninst results in an average overhead of 35%, which is lower than both Dyninst 7.0 (66%) and PIN (90%). The group transformation results in a distinct improvement in the Apache (12% to 0.4%) and MySQL (66% to 51%) benchmarks, but does not have a significant impact on the SPECint benchmarks. This is not surprising, since only position independent code (e.g., library code) includes the `thunk` functions targeted by the group transformation. The Apache and

MySQL benchmarks execute a significant amount of library code, but the SPEC benchmarks do not.

Our poorer performance on two benchmarks (`hmmer` and `h264ref`) is due to the cost of compensating for AVU and CAD sensitivity. Our current implementation uses a simple pointer analysis that over-approximates many pointer accesses as AVU and CAD sensitive; this could be greatly reduced with a more sensitive analysis. This cost is not shared by Dyninst 7.0 or PIN, which do not use such analysis. Dyninst 7.0 does not compensate for either AVU or CAD sensitivity, and PIN does not compensate for AVU sensitivity (as PIN does not modify the original code, there will be no CAD sensitive instructions). This lack of compensation can be exploited by tamper-resistant programs to detect modification and is therefore dangerous. Finally, Dyninst 7.0 failed to correctly run the `omnetpp` benchmark due to an incorrect handling of exceptions; our approach transparently handled this problem since the exception code was determined to be externally sensitive and therefore emulated.

7.7 Summary

We presented a technique for preserving the original behavior of code that was affected by code replacement or instrumentation. Our technique, sensitivity-resistant code relocation, identifies both instructions that are sensitive to code replacement or instrumentation. We then use a slicing-based dataflow analysis to identify which of these sensitive instructions are externally sensitive and will cause the overall program behavior to be altered in a way that is visible in the output of the program. We apply compensatory transformations to such instructions to preserve their original behavior. We evaluated this technique in two ways. First, we applied this technique when instrumenting a set of tamper-resistant binaries created with popular malware packing tools, and demonstrated that sensitivity-resistant code relocation is capable of hiding the alterations made by patch-based code replacement from such binaries. Second, we evaluated the overhead imposed by this approach and showed it was competitive with other current relocation techniques that rely on ad-hoc heuristics to identify sensitivity rather than analysis. In the next chapter, we summarize this dissertation and discuss avenues of future research.

8 Conclusion

Our goal for this research has been to further develop binary modification, the ability to extend or change the behavior of a binary program. In this work, we have been guided by four principles: abstraction, that a user should manipulate high-level representations of the binary rather than binary code directly; timeliness, that a user should be able to manipulate code at any point from pre-execution to while the modified code executes; safety, that the user should not be able to accidentally introduce undesired behavior or create an invalid binary; and efficiency, that the overhead imposed by modification should be proportional frequency with which modified code is executed. In this final chapter, we review our technical contributions and suggest possible directions for future research and areas where we might leverage research in other areas for further refine and improve our techniques.

8.1 Contributions

This dissertation makes the following four main contributions:

CFG-based binary modification We developed the concept of modifying binaries by transforming their CFGs. By operating on the CFG, rather than individual instructions, we allow users to modify binaries without requiring detailed knowledge of the idiosyncrasies of both the binary being modified and the instruction set the binary was written in. We defined CFG validity, which ensures that the transformed CFG can be instantiated in a valid binary. Leveraging this concept of validity, we defined a structured binary editing algebra of CFG transformations that allow detailed modification without creating an invalid binary. Finally, we presented an analysis for determining if a CFG transformation would cause invalid indirect control flow. We demonstrated these techniques were useful by applying them to two tools: a hot-patching tool for applying security fixes to a running Apache web server, and a instruction-level replacement tool, written by Michael Lam, for evaluating when floating-point operations are performed at an unnecessarily high level of precision.

Function and loop abstractions We further developed the concepts of functions and loops, abstractions that encapsulate binary complexity behind familiar interfaces. We defined a efficient, single-entry abstraction for functions that can be applied to highly optimized code across a variety of instruction sets, and reused the Dyninst 7.0 loop abstraction [15]. We then extended our algebra to include function and loop transformations. We demonstrated that these techniques could be used to successfully insert code at function entries and exits in highly optimized code that could not be successfully handled by previous techniques.

Code replacement We extended the concept of patch-based code replacement with two novel techniques, region patching and state interception. Region patching replaces a set of blocks or functions as a single unit, which lowers the overhead imposed by patching. State interception allows us to replace actively executing code by directly modifying process state in addition to using branches. We demonstrated that our approach allows us to modify actively executing code while imposing similar or lower overhead to other methods and preserving the proportional cost expected of patch-based instrumentation.

Code relocation We formalized the effects of code replacement and inserting code upon the behavior of the program and presented a technique for preserving the original behavior of code that was not explicitly modified. This technique allows the execution of a modified program to differ from the original as long as these differences are not visibly apparent in the output of the program. We defined instruction sensitivity to code replacement, and defined an analysis for identifying the externally sensitive instructions that will cause the visible behavior of the program to change. We demonstrated that this code relocation technique results in lower overhead, and allows us to instrument sensitive malware samples that attempt to detect modification.

8.2 Future Directions

We see several opportunities to build upon the ideas we have presented here.

Visual binary editing Our CFG transformations suggest a visual editing idiom for binaries that would allow users to interactively modify the binary, a concept first suggested by the LANCET tool [72]. This idiom would benefit users wishing to perform a manual investigation or modification of a binary; for example, developing a patch or testing an optimization. Visual, interactive editing would require several extensions on this work. First, it seems natural that the user should be able to observe the effects of CFG modification at the instruction layer. This raises questions as to what they should perceive: the raw generated binary, including the effects of code replacement and code relocation, or some abstracted view that hides these details. Second, an expert user may want to directly modify the binary in addition to transforming the CFG. While we did not investigate this area, instead preferring to operate only on the CFG, modifying instructions directly provides a finer granularity of control that may be beneficial. Doing so would require developing a two-way consistency model between instructions and the CFG.

Modifying binaries using other representations We have demonstrated that transforming the CFG is a useful mechanism for altering a program's structure and execution. However, it is not amenable to all forms of binary modification. For example, adding a field to a data structure, while possible to perform on the CFG by individually modifying each instruction that accesses the structure, is more naturally expressed as a transformation of the data dependence graph (DDG). Doing so would also require updating existing process state, such as by the techniques described in [49].

Selecting code regions to replace Our work has shown that replacing code at the granularity of a function or set of functions reduces overhead even when the overhead imposed by relocating more code is taken into account. We believe this is due to the reduced number of interception branches executed by the program, but we have not fully characterized that relationship. For example, it may be possible to obtain lower overhead by replacing a subset of blocks within a function, such as a loop, or replacing additional, unmodified functions. We believe this determination will take three factors into account. First, the structural characteristics of the program, such as the call graph and loops. Second, the cost of code relocation, as discussed in Chapter 7. Third, the processor characteristics, including the cost of executing branches or the size of the cache.

Group compensatory transformations We performed an initial study of transforming a group of instructions during code relocation to further reduce overhead. Our approach was based on the pattern-matching approach used by Dyninst 7.0 [11], which itself was based on manual user examination of binaries. We believe that groups can be derived automatically based on data dependencies instead of identified by hand, and that compiler optimization techniques can be applied to groups to improve efficiency of the generated code.

Other forms of sensitivity We address four classes of instruction sensitivity in Chapter 7: PC-sensitive, CF-sensitive, CAD-sensitive, and AVU-sensitive. Inserted code may alter the behavior of the program in other ways, such as by altering library state, operating system state, the layout of the heap, or the time required to execute; programs that are sensitive to such alterations may have their visible behavior changed. We believe our techniques could be extended to handle these forms of sensitivity. To do so, we would need to specify which instructions would be sensitive to these changes and how their inputs would be changed.

In addition to these new areas, there are several aspects of our work that could be improved by leveraging work in other areas. The foremost of these are our dataflow analyses, both for indirect control flow validity and external sensitivity detection, which would benefit greatly from improved pointer analysis. There are several techniques for thorough pointer analysis of binaries, such as Value Set Analysis [6]; however, these analyses are too computationally complex to be be in the critical path of binary modification. Such analyses could be applied off-line and the results stored, or calculated on a secondary thread of computation that is not on the critical path. Alternatively, we could use a light-weight approximation.

Finally, we believe there are several applications of binary modification that have not been explored simply due to the lack of flexible, easy to use binary modification toolkits. In our personal experience, each improvement we've made has been welcomed by the tools community, who immediately found new applications. We have had discussions with various colleagues who want to apply these techniques in such areas as updating the algorithms used by legacy simulation binaries or testing the resistance of a binary to random memory errors. We hope that this work will inspire both new directions in the area of binary modification and new applications of the technique.

A Branch Overhead

Patch-based code replacement uses interception branches to transfer execution between original and replacement code. These interception branches are frequently performed by unconditional direct branches, such as the IA-32/x86-64 `jmp` or PowerPC `b` instructions. In rarer cases, a sequence that constructs the target address in a register and performs an unconditional indirect branch to that target address is required. We had expected that the first form, direct branches, would have minimal impact on the program's performance due to the optimizing capabilities of the processor. Specifically, we expected the branch prediction unit to elide any fetch penalty involved in frequent branching. However, in our experience this is not the case: frequently branching between two separate code regions enlarges the working set of the program, harms code locality, and introduces overhead due to cache stalls. In this appendix, we discuss our findings, provide detailed performance results, and discuss the impact of branch overhead on binary modification toolkit design.

We first characterize the branching characteristics of Dyninst 7.0 and anytime code replacement. Dyninst 7.0 moves code at two granularities: basic block and function. First, it attempts to relocate each instrumented basic block individually. As a result, Dyninst 7.0 will execute two branches per relocated block: one from the original code to the relocated code at the start of the block, and one from the relocated code to the original code at the end. We refer to this behavior as *ping-pong*. If Dyninst 7.0 encounters a block that is too small to be patched with an interception branch, it will instead relocate the entire function, padding each block in the function with enough space to contain

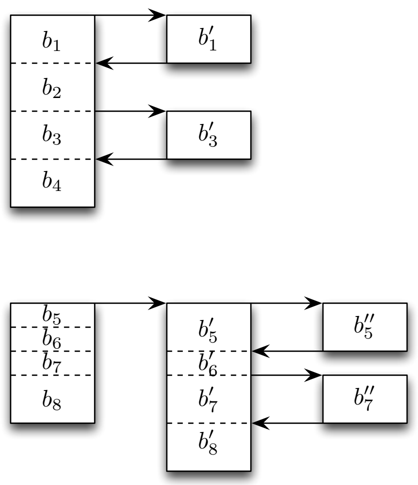


Figure A.1: Example of code patched by Dyninst 7.0, with functions represented by rectangles, blocks by dashed lines within functions, and branches by solid arrows. Dyninst 7.0 may patch either at the basic block level or the function level, as shown. The first function consists of blocks b_1 to b_4, with blocks b_1 and b_3 instrumented. These blocks are each large enough to contain an interception branch and so are patched directly to blocks b_1' and b_3'. The second function consists of blocks b_5 to b_8, with blocks b_5 and b_7 instrumented. These blocks are not large enough to contain interception branches, and so the entire function is relocated first and the instrumented blocks (b_5' and b_7') are padded out to contain additional interception branches to b_5'' and b_7''.

an interception branch. In these cases, Dyninst 7.0 will execute two sets of branches; one at the entry of the function, and then two per block inside the function. We show an example of both block- and function-level relocation in Figure A.1.

Anytime code replacement replaces all instrumented blocks or functions as a single group. Thus, interception branches will only be executed for the following four cases: first, when execution enters the relocated code region from outside the region, such as when a relocated function is called from an original function; second, when execution returns to a block corresponding with a return address on the stack; third, when a patched function is called through a function pointer; fourth, when a patched block is reached with an indirect branch. If the entire binary is rewritten, the first behavior will only occur once. The second does not occur during binary rewriting, or if functions are replaced before they are first executed. The third and fourth behaviors are due to our method of handling indirect control flow; we calculate the original target, branch to that location, and intercept control flow with an interception branch.

Not surprisingly, executing interception branches imposes overhead. However, we would expect this overhead to be small, since a processor should be able to predict the target of the branch and begin fetching instructions from this target with a minimal delay. In practice, this is not the case; reducing the number of branches executed has a significant, not minimal, impact on reducing the overhead imposed by code replacement. We investigated this as follows.

We began by identifying two sample programs to analyze. We selected two programs in the SPEC benchmark suite, `hmmer` and `Xalan`. The `hmmer` program consists of a small kernel and makes infrequent use of indirect control flow. This program exhibits low overhead with anytime code replacement, because execution is contained in the replacement code region, and higher overhead with Dyninst 7.0, because execution returns to original code at the end of each block before being intercepted. The `Xalan` program is larger and makes extremely frequent use of indirect control flow, and thus execution returns to original code and must be intercepted for both anytime code replacement and Dyninst 7.0. By using these two programs, we could distinguish between performance metrics that correlated with poor overall performance and those that did not.

We then measured the execution behavior of each test case. Unlike our previous performance results, these results were gathered from an Intel Core i7 machine running at 2.93GHz; we used this more modern machine because it supported a wider range of hardware performance counters. We used Intel's VTune performance analysis software [30], which uses hardware performance counters to record detailed processor-level behavior in addition to measuring total execution time. Since we believed the overhead was due to additional branches, we focused on the processor's front-end (the ITLB, L1 cache, and decode stages) and execution pipeline (issue, execution, and instruction retirement stages) rather than on the memory subsystem. We show the results of running these analyses on our six test cases in Tables A.1 (for `hmmer`) and A.2 (for `Xalan`).

For `hmmer`, anytime code replacement imposed low total overhead (2%). This is as expected, since `hmmer` does not make use of indirect control flow and thus execution would remain almost entirely in the relocated code region. Interestingly, anytime code replacement actually resulted in a decrease in ITLB misses and L1 stalls. This is a side-effect of our current relocation approach, which removes padding bytes and other unreachable code. As a result, the relocated code is often smaller than the original code. Dyninst 7.0's ping-pong behavior imposed higher total overhead (16%), corresponding with an increase in ITLB misses and L1 stalls; this in turn resulted in an increase in decode, issue, execute, and retire stalls.

For `Xalan`, both approaches imposed higher overhead (146% and 390%, respectively) due to the program's greater use of indirect control flow. These increases again correspond with an increase in ITLB misses and L1 stalls; these misses and stalls also induce decode, issue, execute, and retire stalls in the processor pipeline.

The increase in execution time correlates well with the increased number of ITLB misses and L1 stalls. We hypothesized that the ITLB was under pressure from the additional relocated code. To investigate this behavior, we measured the working set size (in code pages) of each test case. We did so using PIN [41] by recording each unique page that contained executed code; we used PIN since it would record execution in both original code and our relocated code. We show the working set sizes on the final lines of Tables A.1 (for `hmmer`) and A.2 (for `Xalan`).

The data support a hypothesis that branches impose overhead by increasing the working set of the program, which in turn causes poor cache performance. The increase in execution time for each test case is correlated to a higher ITLB miss rate and L1 cache miss rate. These, in turn, create stalls in the processor pipeline at the decode, issue, execute, and retire stages. Since anytime code replacement eliminates ping-pong behavior for direct control flow, it results

	Unmodified	Anytime		Dyninst 7.0	
Execution Time (seconds)	534	544	102%	621	116%
ITLB Misses (millions)	12	9	73%	20	167%
L1 Stalls (millions)	856	832	97%	1,472	172%
Decode Stalls (millions)	346,944	364,794	105%	693,970	200%
Issue Stalls (millions)	344,094	369,252	107%	664,476	193%
Execute Stalls (millions)	153,702	156,996	102%	419,208	273%
Retire Stalls (millions)	422,994	447,666	106%	873,670	207%
Working Set (pages)	221	234	106%	277	125%

Table A.1: Performance results for `hmmer` for an unmodified binary, a binary rewritten with anytime code replacement, and a binary rewritten with Dyninst 7.0. For the unmodified binary we show total times or performance counter values. For the rewritten binaries we show both total times and percentage increases (or decreases) from the unmodified base case. Values are rounded by VTune as a result of its sampling-based approach.

	Unmodified	Anytime		Dyninst 7.0	
Execution Time (seconds)	69.8	102	146%	273	390%
ITLB Misses (millions)	57	298	524%	1,766	3,108%
L1 Stalls (millions)	9,440	35,728	378%	144,448	1,530%
Decode Stalls (millions)	71,196	125,940	177%	464,946	653%
Issue Stalls (millions)	68,130	121,698	179%	460,770	676%
Execute Stalls (millions)	36,696	67,908	185%	360,174	982%
Retire Stalls (millions)	74,766	123,714	165%	439,320	588%
Working Set (pages)	658	984	150%	1,712	260%

Table A.2: Performance results for `Xalan` for an unmodified binary, a binary rewritten with anytime code replacement, and a binary rewritten with Dyninst 7.0. For the unmodified binary we show total times or performance counter values. For the rewritten binaries we show both total times and percentage increases (or decreases) from the unmodified base case. Values are rounded by VTune as a result of its sampling-based approach.

in lower overhead than Dyninst 7.0. Our next avenue of investigation is to also try to eliminate ping-pong behavior for indirect control flow. Several approaches to this problem have been described in the literature; for example, we could update the values used to calculate the indirect control flow target [36] or insert address translation code at the indirect call or branch [10, 41, 52]. This remains an open area of implementation.

Clearly, our initial intuition that branches were inexpensive was incorrect. This impacts the design of a binary modification toolkit, particularly patch-based toolkits. Dyninst 7.0 made the assumption that branches were free, and instead focused on two other capabilities: first, the ability to quickly insert and remove instrumentation by only updating a single basic block, and second, a simple, one-pass code generation design. While Dyninst 7.0 supports these capabilities better than anytime code replacement, it pays the cost in execution overhead of the instrumented binary, and this cost tends to dominate. In contrast, anytime code replacement must regenerate entire regions of code when instrumentation is inserted or removed, and must use multiple iterations to determine control flow targets. Both of these characteristics complicate the design of anytime code replacement and increase the time necessary to generate the replacement code. For example, rewriting `hmmer` took 4.55 seconds with Dyninst 7.0 and 6.55 seconds with anytime code replacement; however, the execution time decreased from 621 seconds to 544 seconds. We believe the performance benefits of reducing branches outweighs the increase in complexity in a binary modification toolkit.

References

[1] Aho, A.V., R. Sethi, and J.D. Ullman. 1986. *Compilers: Principles, techniques, and tools*. Addison-Wesley Publishing.

[2] Andrews, G., S. Debray, and M. Legendre. 2004. BIT user's manual.

[3] Andries, M., G. Engels, A. Habel, B. Hoffmann, H.J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. 1996. Graph transformation for specification and programming. *Science of Computer Programming* 34:1–54.

[4] Apache Project. 2011. Apache httpd daemon.

[5] Bala, V., E. Duesterwald, and S. Banerjia. 2000. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices* 35(5):1–12.

[6] Balakrishnan, G., and T. Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. New York, NY, USA.

[7] Ball, T., and J.R. Larus. 1992. Optimally profiling and tracing programs. In *Nineteenth acm symposium on principles of programming languages*. Albuquerque, NM, USA.

[8] Bernat, A. R., and B. P. Miller. 2011. Anywhere, any time binary instrumentation. In *Program analysis for software tools and engineering (paste)*. Szeged, Hungary.

[9] Bernat, A. R., K. Roundy, and B. P. Miller. 2011. Efficient, sensitivity resistant binary instrumentation. In *International symposium on software testing and analysis (issta)*. Toronto, Canada.

[10] Bruening, D., T. Garnett, and S. Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *First annual international symposium on code generation and optimization*. San Francisco, CA, USA.

[11] Buck, B., and J. Hollingsworth. 2000. An API for runtime code patching. *Journal of High Performance Computing Applications* 14(4):317–329.

[12] Cifuentes, C., and M.V. Emmerik. 1999. Recovery of jump table case statements from binary code. In *International workshop on program comprehension*. Pittsburgh, PA, USA.

[13] Cifuentes, C., and K.J. Gough. 1995. Decompilation of binary programs. *Software - Practice & Experience* 25(7):811–829.

[14] Cmelik, B., and D. Keppel. 1994. Shade: a fast instruction-set simulator for execution profiling. In *Sigmetrics conference on measurement and modeling of computer systems*. New York, NY, USA.

[15] Collins, E. D., and B. P. Miller. 2005. A loop-aware search strategy for automated performance analysis. In *High performance computing and communications (hpcc-05)*. Sorrento, Italy.

[16] Coward, P.D. 1988. Symbolic execution systems-a review. *Software Engineering Journal* 3(6):229–239.

[17] De Bus, B., D. Chanet, B. De Sutter, L. Van Put, and K. De Bosschere. 2004. The design and implementation of FIT: a flexible instrumentation toolkit. In *Program analysis for software tools and engineering (paste)*. Washington, DC, USA.

[18] De Bus, B., B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. 2004. Link-time optimization of ARM binaries. In *Acm conference on languages, compilers, and tools (sigplan/sigbed)*. Washington, D.C.

[19] Desoli, G., N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher. 2002. DELI: A new run-time control point. In *35th annual international symposium on microarchitecture (micro '02)*. Istanbul (not Constantinople), Turkey.

[20] Dimitrov, M., and H. Zhou. 2009. Anomaly-based bug prediction, isolation, and validation: An automated approach for software debugging. In *14th international conference on architectural support for programming languages and operating systems (asplos-xiv)*. Washington, DC, USA.

[21] Drewry, W., and T. Ormandy. 2007. Flayer: exposing application internals. In *Workshop on offensive technologies (woot)*. Boston, MA, USA.

[22] Eustace, A., and A. Srivastava. 1995. ATOM: A flexible interface for building high performance program analysis tools. In *Winter 1995 usenix conference*. New Orleans, LA, USA.

[23] Giffin, J., M. Christodorescu, and L. Kruger. 2005. Strengthening software self-checksumming via self-modifying code. In *21st annual computer security applications conference (acsac)*. Washington, DC, USA.

[24] GNU Compiler Collection. 2012. GNU compiler suite.

[25] Harris, L., and B. P. Miller. 2005. Practical analysis of stripped binary code. In *Workshop on binary instrumentation and applications*. St. Louis, MO, USA.

[26] Heckel, R. 2006. Graph transformation in a nutshell. In *Electronic notes in theoretical computer science*, 187–198. Elsevier.

[27] Higham, N. 2002. *Accuracy and stability of numerical algorithms, second edition*. SIAM Philadelphia.

[28] Hollingsworth, J., and B. P. Miller. 1993. Dynamic control of performance monitoring on large scale parallel systems. In *International conference on supercomputing*. Tokyo, Japan.

[29] Holloway, G., and C. Young. 1997. The flow analysis and transformation libraries of machine SUIF. In *Second suif compiler workshop*. Palo Alto, CA, USA.

[30] Intel. 2012. VTune performance analyzer.

[31] Jackson, D., and E. J. Rollins. 1994. Chopping: A generalization of slicing. Tech. Rep., Carnegie Mellon University, Pittsburgh, PA, USA.

[32] Kahan, W. 1996. The improbability of probabilistic error analyses for numerical computations. Tech. Rep., University of California, Berkeley.

[33] Kiss, A., J. Jasz, G. Lehotai, and T. Gyimothy. 2003. Interprocedural static slicing of binary executables. In *Source code analysis and manipulation*. Amsterdam, The Netherlands.

[34] Krell Institute. 2010. Open SpeedShop.

[35] Lam, M. 2012. Automated floating-point precision analysis. Tech. Rep., University of Maryland, College Park, MD, USA.

[36] Larus, J.R., and E. Schnarr. 1995. EEL: Machine independent executable editing. In *Programming language design and implementation (pldi)*. La Jolla, CA, USA.

[37] Lattner, C., and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization (cgo)*. Palo Alto, CA, USA.

[38] Laurenzano, M., M. Tikir, L. Carrington, and A. Snavely. 2010. PEBIL: Efficient static binary instrumentation for linux. In *Ieee international symposium on performance analysis of systems and software (ispass)*. White Plains, NY, USA.

[39] Lengauer, T., and R.E. Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems* 1(1):121–141.

[40] Loriant, N., M. Ségura-Devillechaise, and J.M. Menaud. 2005. Server protection through dynamic patching. In *Pacific rim international symposium on dependable computing (prdc)*. Changsha, Hunan, China.

[41] Luk, C. K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. PIN: building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation (pldi)*. Chicago, IL, USA.

[42] Maebe, J., M. Ronsse, and K. De Bosschere. 2002. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Pact'02: International conference on parallel architectures and compilation techniques*. Minneapolis, MN, USA.

[43] Miller, B. P., M. Callaghan, J. Cargille, J. Hollingsworth, B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. 1995. The Paradyn parallel performance measurement tool. *IEEE Computer* 28(11):37–46.

[44] Mirgorodskii, A., and B. P. Miller. 2005. Autonomous analysis of interactive systems with self-propelled instrumentation. In *12th annual multimedia computing and networking conference (mmcn)*. San Jose, CA, USA.

[45] Moser, A., C. Kruegel, and E. Kirda. 2007. Exploring multiple execution paths for malware analysis. In *Security and privacy (sp)*. Oakland, CA, USA.

[46] National Vulnerability Database Vulnerability Summary for CVE-2011-3368. 2011.

[47] National Vulnerability Database Vulnerability Summary for CVE-2011-3607. 2011.

[48] National Vulnerability Database Vulnerability Summary for CVE-2012-0021. 2011.

[49] Neamtiu, I., and M. Hicks. 2009. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM conference on programming language design and implementation (pldi)*, 13–24. Dublin, Ireland.

[50] Nethercote, N., and J. Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming language design and implementation (pldi)*. San Diego, CA, USA.

[51] Newsome, J., D. Brumley, D. Song, J. Chamcham, and X. Kovah. 2006. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *13th annual network and distributed systems security symposium*. San Diego, CA, USA.

[52] O'Sullivan, P., K. Anand, A. Kothan, M. Smithson, R. Barua, and A. D. Keromytis. 2011. Retrofitting security in cots software with binary rewriting. In *26th ifip international information security conference (sec)*. Lucerne, Switzerland.

[53] Paradyn Project. 2012. DynC: An instrumentation language for specifying snippets.

[54] ———. 2012. ParseAPI: An application program interface for binary parsing.

[55] ———. 2012. PatchAPI: An application program interface for binary patching.

[56] Reps, T., S. Horwitz, M. Sagiv, and G. Rosay. 1994. Speeding up slicing. In *Symposium on the foundations of software engineering (sigsoft)*. New Orleans, Louisiana, USA.

[57] Romer, T., G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. 1997. Instrumentation and optimization of win32/intel executables using etch. In *Usenix windows nt workshop*. Seattle, WA, USA.

[58] ROSE Compiler Project. 2012. Rose: Making compiler technology accessible to all programmers.

[59] Rosenblum, N., G. Cooksey, and B. P. Miller. 2008. Virtual machine-provided context sensitive page mappings. In *Acm sigplan/sigops international conference on virtual execution environments*. VEE '08, Seattle, WA, USA.

[60] Roundy, K. A., and B. P. Miller. 2010. Hybrid analysis and control of malware binaries. In *Recent advances in intrusion detection (raid)*. Ottawa, Canada.

[61] Roundy, K. R. 2012. Hybrid analysis and control of malicious code. Ph.D. thesis, University of Wisconsin, Madison, WI, USA.

[62] Saxena, P., P. Poosankam, S. McCamant, and D. Song. 2009. Loop-extended symbolic execution on binary programs. In *18th international symposium on software testing and analysis (issta)*. Chicago, IL, USA.

[63] Schwarz, B., S. Debray, and G. Andrews. 2001. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Workshop on binary translation*. Barcelona, Spain.

[64] ———. 2002. Disassembly of executable code revisited. In *Ieee 9th working conference on reverse engineering*. Richmond, VA, USA.

[65] Scott, D., and C. Strachey. 1971. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford University Computing Lab.

[66] Shende, S., and A. D. Malony. 2006. The TAU parallel performance system. *International Journal of High Performance Computing Applications* 20(2):287–311.

[67] Snavely, N., S. Debray, and G. Andrews. 2002. Predicate analysis and if-conversion in an itanium link-time optimizer. In *2nd workshop on epic architectures and compilers*. Istanbul, Turkey.

[68] Srivastava, A., A. Edwards, and H. Vo. 2001. Vulcan: Binary transformation in a distributed environment. Tech. Rep., Microsoft Research.

[69] Srivastava, A., and A. Eustace. 1994. ATOM: A system for building customized program analysis tools. In *Programming language design and implementation (pldi)*. Orlando, FL, USA.

[70] Srivastava, A., and D. W. Wall. 1992. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages* 1(1): 1–18.

[71] Turing, A.M. 1946. Proposals for the development in the mathematics division of an automatic computing engine (ace). Tech. Rep. Report E882, National Physical Laboratory.

[72] Van Put, K., B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere. 2005. Lancet: A nifty code editing tool. In *Program analysis for software tools and engineering (paste)*. Lisbon, Portugal.

[73] Wilkes, M. V., D. J. Wheeler, and S. Gill. 1951. *The preparation of programs for an electronic digital computer*. Addison-Wesley Publishing.

[74] Willems, C., T. Holz, and F. Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. In *Security and privacy (sp)*. Oakland, CA, USA.

[75] Wilson, R., R. French, C. Wilson, Amarasinghe S., J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices* 29.

[76] Wurster, G., P. Van Oorschot, and A. Somayaji. 2005. A generic attack on checksumming-based software tamper resistance. In *Security and privacy (sp)*. Oakland, CA, USA.

[77] Xun, L. 1999. A Linux executable editing library (LEEL).

[78] Zhang, X., N. Gupta, and R. Gupta. 2006. Locating faults through automated predicate switching. In *International conference on software engineering (icse)*. Shanghai, China.

[79] Zhou, J., and G. Vigna. 2004. Detecting attacks that exploit application-logic errors through application-level auditing. In *20th annual computer security applications conference (acsac)*. Tucson, AZ, USA.