ABSTRACT

Title of dissertation: Compilation and Binary Editing for Performance and Security

Tugrul Ince, Doctor of Philosophy, 2013

Dissertation directed by: Professor Jeffrey K. Hollingsworth
Department of Computer Science

Traditionally, execution of a program follows a straight and inflexible path starting from source code, extending through a compiled executable file on disk, and culminating in an executable image in memory. This dissertation enables more flexible programs through new compilation mechanisms and binary editing techniques.

To assist analysis of functions in binaries, a new compilation mechanism generates data representing control flow graphs of each function. These data allow binary analysis tools to identify the boundaries of basic blocks and the types of edges between them without examining individual instructions. A similar compilation mechanism is used to create individually relocatable basic blocks that can be relocated anywhere in memory at runtime to simplify runtime instrumentation.

The concept of generating relocatable program components is also applied at function-level granularity. Through link-time function relocation, unused functions in shared libraries are moved to a section that is not loaded into the memory at

runtime, reducing the memory footprint of these shared libraries. Moreover, function relocation is extended to the runtime where functions are continuously moved to random addresses to thwart system intrusion attacks.

The techniques presented above result in a 74% reduction in binary parsing times as well as an 85% reduction in memory footprint of the code segment of shared libraries, while simplifying instrumentation of binary code. The techniques also provide a way to make return-oriented programming attacks virtually impossible to succeed.

Compilation and Binary Editing for Performance and Security

by

Tugrul Ince

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair/Advisor
Associate Professor Ahmet Aydilek, Dean's Representative
Professor Ashok Agrawala
Associate Professor Atif Memon
Professor Alan L. Sussman

© Copyright by
Tugrul Ince
2013

To my lovely wife Şimal,

To my parents Feriz and Halime,

And to the rest of my family:

Mehmet, Şükran, Kürşat, Nurgül, Çağrı, Hilal, and Işıl.

Acknowledgments

I want to first thank my advisor Prof. Jeffrey K. Hollingsworth for all the support and guidance he has provided. Thanks to him, I now feel prepared to take on challenges that lie ahead.

I also want to extend my gratitude to all members of my dissertation committee for finding my work interesting enough to sit through a presentation on a beautiful June day through lunch.

I was lucky enough to meet some great people along the way. A big thank you to all of you guys. A.T., you taught me how to be passionate about research. Geoff, you showed me mental and physical fitness can go hand-in-hand, and made me think outside the box (and the bun). Nick, you kept showing me what to expect two years down the road, and that things turn out OK in the end. Mike, you kept me up-to-date with everything going around and with all the floating-point 'stuff'. Ray, you taught me '*ls*' and how to enjoy good food. Things I will remember the most are the long-gone public phone in front of the A.V. Williams, the wind tunnel effect near the University View, the long list of movies I have to watch at one point, the graphs passed around on Fridays, and that a four-door car is a luxury.

I also met many wonderful people who, for some reason, do not enjoy trying to make sense of 1's and 0's as much as I do. A big shout out to Ali Fuad, Barış, Bedrettin, Bengü, Burcu, Çağdaş, Elif, Evren, Fazıl, Ferhan, Füsun, Günay, Okan, Rezarta, Serap, Tıkır, Tuğba, and Uğur. You all left your marks on this document.

I owe many thanks to my family. I can never pay you back for all the support you have been providing all these years. Dad, I know you would be proud. Mom, Kürşat, and Çağrı, thank you for supporting me in pursuing my goals. A big thank you to my in-laws Mehmet, Şükran, Nurgül, Hilal, and Işıl. This world would be quite boring without you.

Finally, I want to thank my wife, Şimal, for always being there for me. Without you I would have long lost my way along the treacherous hallways of graduate school. You held my hand and pulled me to safety. Thank you for having the most amazing dimples in the world.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

2 Related Work 7
- 2.1 Compilation Techniques for Binary Analysis and Debugging 7
- 2.2 Binary Analysis and Editing Tools 9
- 2.3 Binary Parsing 12
- 2.4 Memory Footprint Minimization 13
- 2.5 Randomization for Security 15

3 Instrumentation with Relocatable Basic Blocks 21
- 3.1 Overview 21
- 3.2 Relocatable Basic Blocks 22
- 3.3 Compilation 26
- 3.4 Instrumentation 27
- 3.5 Benefits 29
- 3.6 Experimental Results 30
- 3.7 Discussion 35
- 3.8 Conclusion 35

4 Compiler Help for Binary Analysis Tools 37
- 4.1 Overview 37
- 4.2 Difficulties of Binary Parsing 39
- 4.3 Compiler Help 41
  - 4.3.1 Basic Block and Edge Tables 42
  - 4.3.2 Compilation Process 44
  - 4.3.3 Reconstruction 48
- 4.4 Evaluation 50
  - 4.4.1 Environment 51
  - 4.4.2 Experimental Results 52
    - 4.4.2.1 Experimental Parsing Results 53
    - 4.4.2.2 Build Time Metrics 55
    - 4.4.2.3 Running Time Metrics 57
- 4.5 Discussion 58
- 4.6 Conclusion 60

5 Profile-driven Selective Program Loading 61
5.1 Overview 61
5.2 Architecture 63
5.3 Target Applications and Platforms 64
5.4 System Design 64
5.4.1 Profiling 65
5.4.2 Rewriting 66
5.4.2.1 Avoiding Loading Unused Functions 66
5.4.2.2 Update Relative Calls 68
5.4.2.3 Update Symbols 68
5.4.2.4 Update Jump Tables 69
5.4.2.5 Update Function Address Transfers 70
5.4.3 On-demand Mapping 71
5.5 Experimental Results 71
5.5.1 Environment 72
5.5.2 Results 72
5.6 Conclusion 77

6 Security through Runtime Function Relocation 79
6.1 Overview 79
6.2 NINJA: Runtime Function Relocator 81
6.3 Relocatable Functions 82
6.3.1 Function Calls and Function Table 84
6.3.2 Initial Function Location 85
6.3.3 Indirect Branches That Use Tables 86
6.3.4 Accessing Parameters Stored on Stack 87
6.4 Runtime Function Relocation 87
6.4.1 Relocation Strategies 89
6.4.2 Memory Management 90
6.5 Security Implications of NINJA 91
6.6 Security Evaluation 95
6.7 Model of NINJA Security 98
6.8 Performance Evaluation 102
6.9 Conclusion 105

7 Future Work 107
7.1 Short-term Road Map 107
7.1.1 Compile-time Support for Function Relocation 107
7.1.2 Compile-time Support for Reducing Memory Overhead 108
7.1.3 Secure Portable Devices 108
7.1.4 Analyzing Effects of Function Relocation to ICache and TLB . 109
7.1.5 Post-link-time Function Inlining 109
7.2 Long-term Road Map 110
7.2.1 Live Virtual Machine Migration 110
7.2.2 Decoupling Execution from Physical Mediums 111

7.2.3 Use of Hardware Performance Counters for Runtime Code Modification 112

8 Conclusion 115

Bibliography 119

List of Tables

3.1 Relocatable Basic Block Generation Operations on Edges 25

4.1 Properties of Executables Built with Basic Block and Edge Tables . . 55
4.2 Firefox Performance with Tables using V8 JavaScript benchmark . . . 58

5.1 Program Loading: PETSc Results for snes Package 73
5.2 Program Loading: PETSc Results for ksp Package 74
5.3 Program Loading: GS2 Results 75

6.1 Parts of Functions Modified by NINJA 83
6.2 Time Required for a Successful Attack with/without Runtime Function Relocation 97
6.3 Runtime Overhead of Runtime Function Relocation on SPEC CINT 2006 104

List of Figures

3.1 Relocatable Basic Block Creation Techniques 24
3.2 Relocatable Basic Block Compilation Mechanism 26
3.3 Runtime Patching with Relocatable Basic Blocks 28
3.4 Running Time Comparison of Relocatable Basic Block Techniques . . 31
3.5 Normalized Running Times with Relocatable Basic Blocks 32
3.6 Running Times of Instrumented Binaries with Relocatable Basic Blocks 33
3.7 Instruction Counts with Relocatable Basic Blocks 34

4.1 A sample CFG and associated Basic Block and Edge Tables 41
4.2 Formats of Basic Block and Edge Tables 42
4.3 Compilation Mechanism for Basic Block and Edge Table Creation . . 45
4.4 Regular vs. Augmented Assembly Files 47
4.5 Interaction between Compilers and Binary Analysis Tools 49
4.6 SPEC CINT2006 Benchmarks: Normalized Parsing Times 53
4.7 PETSc Example Applications: Normalized Parsing Times 54
4.8 Firefox Executables: Normalized Parsing Times 55

5.1 Overview of the Selective Program Loading System 63
5.2 Running Times with Selective Program Loading 75

6.1 Overview of Runtime Function Relocation System 81
6.2 Non-patched Linux Kernels Provide Only 12-16 bits of Entropy . . . 93
6.3 Reduced Entropy on Little-endian Architectures 94
6.4 Probability of a Successful Attack with ASLR and NINJA 97
6.5 Likelihood of a Successful Attack in 24 Hours 101
6.6 Duration a Function Occupies the Same Address 102
6.7 Probability of a Successful Attack with Varying Relocation Parameters 103
6.8 Relocation Overhead with Varying Relocation Frequencies 105

Chapter 1

Introduction

The evolution in computing has made it possible to run programs at consistently increasing speeds. We are now at a point where contemporary supercomputers can execute programs at a rate higher than 10 petaFLOPS. This execution speed is achieved by carefully optimizing underlying infrastructure such as the operating system, executable file format, and even the Instruction Set Architecture (ISA); resulting in tightly packed code that is free from redundancies.

Consequently, executables tend to be very strict about their layout and execution environment. Linkers position every piece of code at a specific location inside a module. Their offsets are fixed throughout the lifetime of the executable. After a program is fully-compiled, it cannot be modified without extensive operations on the binary, since code is only meaningful at its assigned location. During execution, the loader places each associated module into its assigned memory location. Once a module is loaded, its address is fixed throughout the execution. Even parts of relocatable code that do not have absolute locations before execution are mapped to fixed locations at launch-time and cannot be moved afterwards. While some execution environments such as the Java Virtual Machine provide some level of flexibility in this process such as generating machine code just-in-time, others are very restrictive about this compile-link-load-execute cycle.

Quite often, a program needs to be modified and patched even after the compilation phase is long over. Such cases sometimes arise due to the need to maintain legacy programs whose source code is not available any more. Other times, binary code has to be modified at runtime to apply security patches, or to instrument execution at various points such as function entry and exit points. When a basic block in a function is instrumented, it needs to be relocated. This process involves updating addresses inside that basic block, inserting a branch instruction at the original location, and adding adding a branch instruction to the end of that basic block. As a result, instrumenting a basic block is error-prone and damages the performance.

Runtime instrumentation is not the first nor the only tedious task that involves strict program layout. Before we can do anything with the binary in hand, we need to analyze the code and identify various parts such as functions and basic blocks inside these functions. These structures are recognized after a thorough analysis of the binary, examining it instruction-by-instruction, a process called binary parsing. Any operation with binaries turns out to be extremely costly due to this instruction-by-instruction parsing operation.

Strict layout of code also has drawbacks in memory usage. If demand paging is not available, programs have to be loaded into the memory at launch time along with the shared libraries they use to enforce that all functions that may be used during execution are available in the memory at runtime. Some of these functions may never be called, yet they still occupy memory. In addition to the unnecessary memory overhead, these functions also damage the instruction cache since they also potentially use up precious instruction cache space.

The fact that code has to live at fixed locations also exposes serious security risks. Many remote attacks benefit from injecting malicious code at critical locations in the memory space of the target process. The location of critical code is easy to guess when it has to live at a fixed location. Off-the-shelf and open-source software are especially vulnerable: When the attacker can obtain an exact copy of the target software, finding addresses of critical functions is much easier. Even shared libraries tend to be loaded at the same address across multiple executions. After locating a target function, a typical *return-oriented* attack can be carried out by exploiting a buffer-overflow to make a vulnerable program execute this target function.

We believe these restrictions, especially the fact that code has to reside at a predetermined location throughout its existence, are overly-conservative. To provide more flexibility to the users as well as developers, code should be usable wherever it is loaded. That brings us to the thesis of this work:

Relocatable program components, generated by binary rewriting of programs, results in executables with enhanced security and more efficient memory usage.

We make the following contributions in this dissertation:

We introduce a system that generates fully-relocatable functions:

Correct execution of several types of instructions in executables depends on their locations: Branches require the target code to be present at its expected address, table-based branches read their targets from a branch table that stores addresses of the target code, call instructions include a target address where the callee should reside, etc. Even shared libraries that are compiled into position-independent code (PIC), and hence are considered '*relocatable*', are only relocatable at the granularity of whole libraries and any smaller chunk of code cannot be relocated by itself. The reason is that any call instruction that targets a given function will not work if that target function is moved to an arbitrary location. Therefore, position-independent code alone is not enough for arbitrary function relocation. A function can only be relocatable if it is decoupled from the location it occupies. In this work we present a way to convert any executable to be fully relocatable at the function granularity, along with a technique for self-relocation of functions at runtime without requiring user input.

We provide an execution environment that is more resistant against malicious attacks: Relocating the critical pieces of code provides resistance against malicious attacks as attackers cannot easily determine the correct location of a function they want to execute. As part of my dissertation, we develop a system that continuously relocates functions at runtime so that attackers cannot identify functions they want to execute forcibly. Our mechanism first converts binaries to include fully relocatable functions, then allows programs to relocate their functions at runtime. A combination of relocation strategies provides a high level of randomization while limiting the runtime overhead introduced by the relocation operations. Programs that continuously relocate their functions benefit from a more secure execution environment.

We decrease the memory footprint of shared libraries through profiling and selective program loading: We created a system that prevents unused functions from being loaded into the memory. Moreover, we grouped them so that they also do not occupy space in the instruction cache. In the unlikely event of a call to one of these unused functions, our system dynamically loads the target function and satisfies the call operation. As a result, the overall memory footprint of application drops drastically without incurring any performance overhead.

We introduce relocatability at the basic block granularity for easier instrumentation: Another concept we introduce in this work is *relocatable basic blocks*. They are used to simplify binary instrumentation process. Relocatable basic blocks can be moved without having to deal with other basic blocks around them. Similarly, we do not need to insert a branch instruction in the old location after the relocation, removing the need to use traps when relocated basic blocks are small in size. As a result, tool builders can create simpler and more manageable instrumentation tools.

We improve binary analysis speeds through faster binary parsing with the help of a new compilation infrastructure: As part of this work, we create a new mechanism to aid with the parsing of binaries. We store information about the locations of basic blocks in a table inside the executable file with the help of a compiler extension. During parsing, any binary analysis tool can read this table and identify functions and basic blocks bypassing the need to decode every instruction in the binary. This mechanism considerably reduces binary parsing time and improves the instrumentation experience.

We show that the above goals can be met with no performance drawback in some cases, and with tolerable slowdown in others: We concentrated our efforts to optimize the runtime performance of applications while providing these important functionalities. Our systems that deal with avoiding loading unused functions and that generate tables to improve binary parsing do not introduce any runtime overhead. Our runtime relocation operations and relocatable basic block approach introduce only limited overhead.

Chapter 2

Related Work

In this chapter, we cover the related work in the area of binary analysis and editing for security and optimization. We first cover some compiler features for executable file generation. Then we cover earlier work in the area of binary analysis and binary editing tools. Binary parsing techniques and randomization strategies for increased security are discussed next. Finally, we present previous work in the area of reducing code size and the memory footprint of programs.

2.1 Compilation Techniques for Binary Analysis and Debugging

A compiler's most important task is to generate executable code. However, compilers are also analysis tools that generate a wealth of information about the program being created. Contemporary compilers store some of this information inside binary files in the form of debug information. Such compilers include the GNU Compiler Collection [34], Intel C++ Compiler [41], The Portland Group compilers [78], LLVM's Clang [53], Oracle's Java compiler [66], and Microsoft Visual Studio compilers [62]. Debug information consists of a map from line numbers in source code to addresses in binary or intermediate languages, explanation of data types, function signatures, locations and types of variables, etc., and is either stored in a separate file (e.g. PDB) or is embedded into executable files (e.g. DWARF, Java class files). Along with the debug information, compilers also emit a list of symbols to show locations of functions and global variables while generating native code.

There are other data structures, also generated by compilers, that are vital to the correct execution of programs: Exception handling tables contain a mapping from instruction addresses to exception handling routines; unwind tables are generated to assist with call chain and stack analysis, and are instrumental during exception handling; and virtual function tables are used in object-oriented programming languages.

None of the compilers we analyzed have support for generating runtime relocatable functions; or marking basic block boundaries and the relationship between them to assist with the analysis of binaries. The compiler feature that is the most relevant to the work presented in this dissertation is the just-in-time compilation of code, a technique used by the Java Runtime Environment and the .NET Framework. In this technique, program code is stored in a position-independent way in an intermediate language (Java Bytecode or Common Intermediate Language). At runtime, the just-in-time compiler may choose to convert this intermediate representation to the native code, positioning the resulting code anywhere in the memory. After the initial compilation of code, the just-in-time compiler can invalidate those functions and recompile them to occupy a new address.

Another compiler feature that is relevant to the work presented here is header pre-compilation, a feature supported by all the compilers mentioned here that generate native code. Unlike the techniques described in this document to help with the analysis of programs, header pre-compilation is used to assist with the initial compilation of the source code.

Later in this work, a mechanism to improve the memory footprint of shared libraries based on profiling is discussed. Compilers use a similar mechanism to optimize the hot paths in a program using a feedback loop. Programs are profiled after the initial compilation. The profiling data is then fed back to the compiler to optimize the instruction cache through function reordering, and to align basic blocks on the hot path to avoid branching for a streamlined execution. Compilers can also use this profiling data to emit code that makes the runtime loader skip loading unused functions into the memory; however, this feature is not yet supported by any of the compilers listed above.

2.2 Binary Analysis and Editing Tools

Analyzing programs is necessary for understanding the program code and improving its quality. However, not all program analysis is performed at the source code level. Analyzing binary code is crucial for performance analysis and identifying malicious software. Since machine code is not easily understandable by software developers, special tools are needed to analyze binary programs. In this section, we cover several such tools.

Dyninst is a runtime code patching library [11]. It provides an interface for tool developers to inject instrumentation code into a program while it is running. Users create their instrumentation using Dyninst's C++ application programming interface (DyninstAPI). This instrumentation code is injected into the target program at runtime. In addition to runtime instrumentation, Dyninst provides useful abstractions for modules, functions, control flow graphs, basic blocks, instructions, and so on. These abstractions are accessible even if the application is not launched. Hence, Dyninst can also be used for static analysis of binary programs. Moreover, using Dyninst's rewriting functionality, the binary can be modified and then can be rewritten to the disk so that the modified binary behaves differently from the original binary.

ATOM is another dynamic instrumentation tool that injects code into a running program and extracts information requested by the end user [76]. Its functionalities are very similar to those of Dyninst. One difference between ATOM and Dyninst is that, unlike Dyninst, instrumentation code in ATOM and original code use two separate copies of the same function if they both need to use it. ATOM provides two methods for memory allocation by the instrumentation code. One of these methods preserves the locations of heap allocated data structures used by the original program, but it is inefficient. The more efficient method intermingles heap allocated memory by the original program and the instrumentation code. In this second approach, data structures allocated by the original program may be placed to distinct locations every time the program is run.

Vulcan is a dynamic instrumentation and binary rewriter tool from Microsoft Research used for optimizations for code locality, procedure inlining, and cross-component optimizations [75]. It is available on multiple architectures and provide many useful abstractions such as System, Program, Component, Procedure, Basic Block, and Instruction. This tool has been used to merge Dynamic-Link Libraries (i.e. DLLs), to apply aggressive cross-component optimizations, and to realign branches in binaries. Vulcan-instrumented binaries can also be rewritten to the disk.

Pin is an instrumentation tool developed by Intel [59]. Like other instrumentation tools, it provides an interface to add code to a process at runtime. The difference in Pin's approach is that it applies just-in-time recompilation to the whole executable. Whenever a new sequence of code is accessed, Pin compiles a copy of that sequence along with all the instrumentation code and executes that copy. Pin handles branching by intercepting taken branches, generating code for the branch target, and executing this newly generated code. To reduce the performance penalty of this approach, Pin performs optimizations such as trace linking (avoiding intercepting code for known branch targets), inlining, and liveness analysis.

Another well-known instrumentation tool is Valgrind [65]. Like Pin, it executes target programs using just-in-time compilation. Valgrind, too, provides various abstractions and an intermediate language to hide platform-specific details from end users. It is extensible through plugins called *skin*s. One drawback of Valgrind is that it runs target program on a simulated CPU hence it does not provide a high execution speed. Besides, Valgrind has trouble working with self-modifying code.

Other dynamic instrumentation tools and rewriters include Shade [15], DynamoRIO [9], EEL [52], and DELI [24].

2.3 Binary Parsing

Parsing binary code has been studied extensively in the past. Since parsing binaries usually requires disassembling machine code, most earlier work focuses on this aspect of binary analysis. Several researchers created higher level representations of machine code following the disassembly. Examples of this approach include Cifuentes and Gough with their decompiler, dcc [14], and Emmerik and Waddington with their Boomerang-based decompiler [28]. More recent work concentrates on disassembly of obfuscated code to identify malicious software [49, 80]. Some researchers, such as Aaraj et al. [1] and Guo et al. [36], combine static disassembly techniques with dynamic analysis to cope with malware. Similarly, Bruschi et al. attempt to identify malware by building a CFG from binary code and comparing it with those of the known malware [10]. Disassembly techniques also made their way into the mainstream applications: Many common tools such as *gdb*, *objdump*, and *IDA* (formerly known as *IDA Pro*) [39] generate disassembly of binary files. The work in Chapter 4 specifically targets applications where the source code is available, so code obfuscation is not an issue for our parsing efforts.

Many tools build CFGs once the executable file is disassembled. De Sutter's [20] and Theiling's [79] control flow generation algorithms perform disassembly of machine code followed by building basic blocks and finally CFGs using the disassembled instructions. Cifuentes et al. [13] and Kiss et al. [45] perform intra- and inter-procedural static slicing on binary files respectively, following disassembly and CFG generation.

All these systems make use of the debugging symbols whenever possible. Many tools also perform a best-effort approach to identify function locations if the symbols are not present. In one example, Harris and Miller demonstrate their tool that finds and disassembles functions with a model that supports multiple entry points for each function [37]. In another, Rosenblum et al. combine common recursive disassembly of machine code with machine learning techniques to identify functions within the gaps between known functions [70].

2.4 Memory Footprint Minimization

Reducing the memory footprint of programs has been extensively researched. Earlier work includes a wide range of techniques from code compression [16, 21, 84, 55] to procedure abstraction [46, 47, 22] and dead code elimination [22].

Code compression is the act of reducing the size of program code by its equivalent representation in another form [5]. It is usually applied to executables that run on embedded systems. Xie et al. developed a system where only the instructions that are the least frequently used are compressed [84]. Just like we do in Chapter 5, they first profile the executable and identify the regions that are the least likely to be used. These regions are then compressed. They leave frequently accessed regions uncompressed to reduce the performance hit. A decompressor generates the original uncompressed code if a block of code that was compressed is accessed at runtime. Lefurgy et al. evaluate a hardware assisted code compression system from IBM PowerPC 405 [55]. In this system all program code is compressed. They note that they achieve performance increase in many situations thanks to faster prefetching of instructions. Since their system relies on CodePack hardware support available on PowerPC 405, it is restricted to this platform. Other approaches to code size reduction techniques include dead and redundant code elimination, procedure abstraction, and instruction level modifications [22, 23]. These methods are demonstrated in the binary-rewriting tool *squeeze* along with interprocedural constant propagation and strength reduction. Developers of *squeeze* also perform procedure extraction for single entry-single exit sections at the binary level. Moreover, they make use of various optimizations such as instruction reordering and platform specific improvements such as reducing the cost of function prologues and epilogues. Van Put et al. propose optimizations including constant propagation and unreachable code elimination as well as procedure extraction in their binary rewriter tool, DIABLO [81]. They also demonstrate how their system can be used to rewrite Linux kernel for specific embedded systems. Komondoor and Horvitz propose procedure extraction at the source code level [46, 47]. Zmily and Kozyrakis propose BLISS which successfully targets reducing text space, energy use and execution time [87]. They selectively replace 32-bit instructions with 16-bit instructions. Since more instructions fit into the instruction cache, performance of the system increases. They also remove repeated sequences of instructions leaving a single copy, just like procedure extraction. Lau et al. show how echo instructions can be used to remove duplicates of identical or similar regions of code [54]. *'echo'* is a proposed instruction that directs processor to execute a sequence of instructions in the binary. Their proposed system performs procedural abstraction and replaces similar sections of code with a single echo instruction.

Zhang and Krintz propose a system that unloads code regions from a modified Java Virtual Machine after their execution is over [86]. They note that 61% of code is only used at the start-up period and can be unloaded after their execution. Although the system we describe in Chapter 5 currently does not unload code regions once they are loaded, this functionality is a straightforward extension to our system.

One work that improves cache utilization and performance of the paging system is carried out by Pettis and Hansen [68]. They present two strategies by carefully repositioning code using execution profiles. The first strategy they employ is grouping functions so that callers and callees are placed close to each other. The second strategy is moving basic blocks inside a function using profiling data so that the execution is streamlined and does not involve many jumps for the common case. Basic blocks that are never used during training runs are then moved to the end of the file as if they were a separate function. The main motivation behind this rearrangement is to reduce the number of taken branches to help branch predictors. Calder and Grunwald improved the Pettis and Hansen algorithm by adding a cost model that reflects architectural properties [12].

2.5 Randomization for Security

In Chapter 1, we mentioned relocating whole programs or their parts (i.e. *randomization*) provided resistance against malicious attacks. Randomization techniques have been around for about 10 years now. In this section, we cover some of the well-known techniques for program randomization.

The PaX Team were the first to introduce Address Space Layout Randomization (ASLR) [67], which has become the *de facto* standard for randomization. The mechanism they developed loads segments of applications into random locations in the memory. Since attackers cannot easily guess where critical sections or functions are located after this randomization, the likelihood of success diminishes. However, ASLR has many limitations some of which are covered in Chapter 6. Xu et al. worked on a similar system where randomization is provided by a modified dynamic program loader with their Transparent Runtime Randomization (TRR) work [85]. Besides program code, their system also relocates the Global Offset Table (GOT) during program launch to provide more resistance against attacks. On Windows, address space randomization was introduced by Li et al. [57]. Giuffrida et al. developed a mechanism to enable address space randomization for the operating system kernel [33]. Their approach uses a background process which periodically randomizes OS components stored in LLVM bitcode file format[1].

Address space randomization mechanism has also been applied to mobile devices. Bojinov et al. introduced an ASLR-like system for Android devices [7]. Since the Android OS prelinks shared libraries, applying address space randomization to shared libraries at launch time does not work. On these devices shared libraries are randomized every time prelinking is performed (i.e. during system updates). Since then, Android 4.0 (Ice Cream Sandwich) introduced an early form of ASLR followed by the implementation of full ASLR on Android 4.1 (Jelly Bean) [60].

[1]LLVM converts source code into an intermediate representation stored in its proprietary '*bitcode*' file format.

Shacham et al. evaluated the effectiveness of address space randomization techniques on 32-bit and 64-bit architectures [72]. They showed that brute force attacks are a big concern against services that fork many child processes. Since each forked child retains the same memory layout as the parent process the attacker has practically an infinite number of processes to attack. This work also demonstrated that the number of possible layouts on a 32-bit machine is only 2^{16}, making an attack to succeed only in 216 seconds on average.

Randomization can also be applied to data. Kil et al. provided a system called Data Space Randomization that can relocate stack, heap, and memory mapped regions [44]. They also randomized program code as in ASLR. However, they need specific relocation information attached to each binary file which is not available by-default, forcing a recompilation of programs with specific compiler flags. Another work that focuses on data space randomization was conducted by Bhatkar and Sekar [6]. Their system relies on keeping data fields encrypted until they are used. Even if attackers can access to critical memory locations, any data they write is considered junk as it cannot be decrypted correctly. Lin et al. introduced a different approach where stack variables and data fields in structures and classes are randomized [58]. During compilation these fields and variables are reordered, and junk data are inserted between each field and variable to increase the power of the obfuscation. However, distributing such software is problematic as the software has to be compiled separately for each customer to provide more randomization.

Another defense approach is using canary values around the return addresses stored on stack. StackGuard has been the first work to offer protection for the return addresses [17]. It was later extended by Etoh and Yoda with Propolice that reorders the stack frame and prevents buffer overflows from overwriting return addresses [29]. Wang et al. used a taint-based detection mechanism to identify cases where the return address is overwritten [82].

Yet another randomization technique for security is the Instruction-Set Randomization [42]. In this approach, instructions that will be executed are stored in an encrypted form and are only decrypted by the processor right before execution. Any code injected by an attacker will, therefore, be invalid as it will not be encrypted properly and will cause a crash. However, this approach requires a processor that can decrypt instructions before executing them, and to date no general purpose processors have been built with this feature. Boyd et al. extended this work to interpreted languages and SQL [8]. Their system encrypts all keywords in the target language. When a keyword that is not encrypted correctly is encountered, it is marked as malicious and is rejected by the system preventing any damage it could cause.

Antonatos et al. extended the randomization idea to network addresses to prevent hit-list attacks[2] with their Network Address Space Randomization (NASR) work [3]. They indicated that services should obtain a new IP-Address periodically so that when a hit-list attack is launched, the target will not be located at the specified IP-Address and, therefore, will be saved.

Wartell et al.'s work on *Binary Stirring* is similar to our work in the sense that

[2]To increase the speed of infection, some worms attack a list of potentially vulnerable machines before employing random attacks. This attack style is named *hit-list attack*.

they also employ self-randomization of binary code [83]. However, the randomization takes place only once at launch-time, leaving the executables vulnerable when they spawn new children, just like the ASLR. Another similar work to ours is the recent work by Curtsinger and Berger [18]. Their self-relocation system moves functions at runtime to perform consistent performance evaluation. Since the main purpose of this work is to create a performance analysis of applications, and not to introduce randomization, relocated functions are still accessible from their initial locations. Clearly, this approach will not enhance security. Moreover, unlike our work, this system requires a recompilation of applications.

Chapter 3

Instrumentation with Relocatable Basic Blocks

This chapter explains our efforts in creating individually relocatable basic blocks. First, we discuss ways to make basic blocks relocatable. Then we talk about how our compilation mechanism can create these basic blocks. We continue to explain how these basic blocks are instrumented, and the benefits of using these basic blocks. We end with our running time experiments and a discussion of the techniques described in this chapter.

3.1 Overview

A basic block is a region of code that has a single entry and a single exit point. Execution of a basic block can only start from the first instruction in the basic block; there cannot be any jump targets inside a basic block except the very first instruction. All instructions in a basic block are guaranteed to execute when the first instruction is executed, as long as there are no exceptions.

The fact that all instructions in a basic block get executed either altogether or not at all makes basic blocks a natural abstraction for many analyses and data structures. Researchers have repeatedly selected basic block granularity in their work. Examples include *Control Flow Graphs* where nodes represent basic blocks, and rearrangement of basic blocks to minimize branching. In this chapter, we also use basic block granularity to generate relocatable code segments.

We believe programs should be as flexible as possible. Code should execute correctly wherever it is located. This flexibility can only be satisfied if the code is made relocatable. Relocatable code generation has been supported to some extent by various compilers for decades. Shared library code is compiled into position independent code (PIC) since a shared library can be loaded anywhere in the memory. In this chapter we show benefits of making individual basic blocks relocatable. Our system generates relocatable basic blocks not just for shared libraries but also for statically linked executable files.

Relocating a regular basic block is not a straightforward process: if a basic block moves, addresses that are used inside that basic block and addresses that refer to it have to be modified with respect to the new location of that basic block. Clearly, these modifications are unaffordable at runtime. In the following sections we first describe how we generate relocatable basic blocks and then discuss the performance of various techniques we investigated.

3.2 Relocatable Basic Blocks

A basic block can only be relocated if the code it contains is not position-dependent. Therefore, any file that needs to be relocatable has to be compiled into position-independent code. Binary files compiled into position-independent code are only relocatable as a whole and any particular basic block in these files is not relocatable by itself. Branch instructions refer to other basic blocks and contain some form of addressing even in position-independent code. If either the branch instruction or the branch target is relocated, the target for that branch instruction will not be computed correctly at runtime. Therefore, branch instructions in our relocatable basic blocks do not contain addresses of targets. Instead, target addresses are looked up at runtime. In our approach, addresses of basic blocks are stored in a table. Branch instructions use this table to identify the locations of target basic blocks.

We investigate three techniques to create relocatable basic blocks, as seen in Figure 3.1. All of these techniques involve a table called *Basic-block Linkage Table* (BLT) that is responsible for directing execution to the target basic block. This indirection introduces some level of runtime overhead. Our three approaches represent different points in the design space of trading ease of relocation and performance.

The first technique we tried makes branch instructions jump to the BLT, as seen in Figure 3.1(b). Entries in the BLT contain branch instructions themselves. These branch instructions read target addresses from another table called *Target Address Table* (TAT). Branch instructions in the BLT can then transfer execution to the correct target. We call this technique *BLT with TAT*.

The second technique avoids one level of indirection by using the BLT to store target addresses and eliminating the need for the *Target Address Table* (Figure 3.1(c)). Each branch instruction jumps to the BLT which in turn jumps to the target address encoded inside BLT. This technique is as powerful as the first technique with better runtime performance, but addresses are intermingled with jump instructions. This technique is called *BLT Only*.

The third technique, as seen in Figure 3.1(d), benefits from fall-through edges

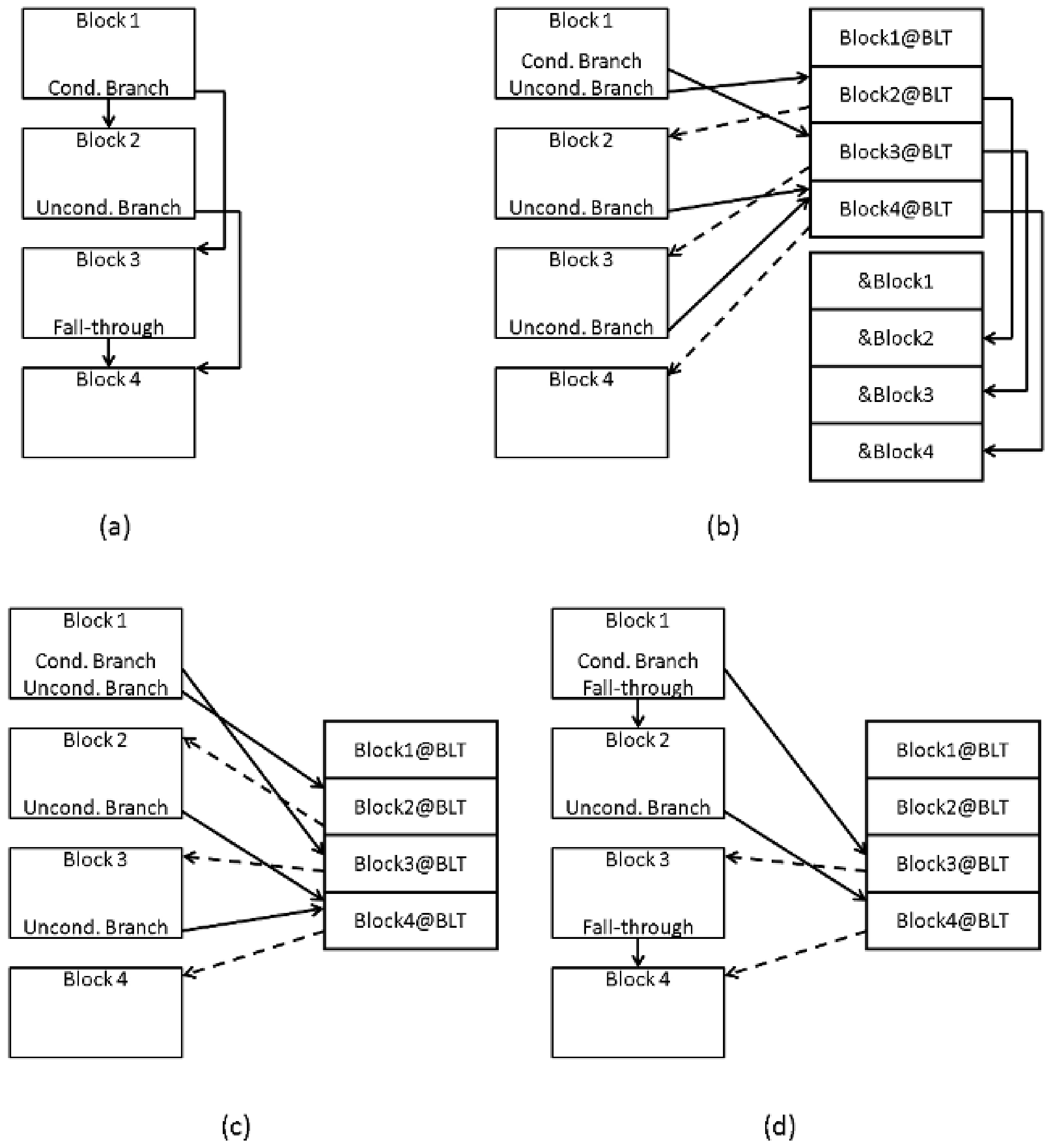


Figure 3.1: Relocatable Basic Block Creation Techniques: (a) Unmodified branching. (b) Basic Block Linkage Table (BLT) with Target Address Table (TAT). (c) Basic Block Linkage Table Only. (d) Basic Block Linkage Table with Fall-Through

Table 3.1: Relocatable Basic Block Generation Operations on Edges

Name	**Fall-through or Call**	**Unconditional Branch**	**Conditional Branch**
BLT with TAT	An unconditional branch to BLT that reads target from TAT	An unconditional branch to BLT that reads target from TAT	A conditional branch for branch target & an unconditional branch for fall-through both to BLT that reads targets from TAT
BLT Only	An unconditional branch to BLT	An unconditional branch to BLT	A conditional branch for branch target & an unconditional branch for fall-through both to BLT
BLT with FT	No modification	An unconditional branch to BLT	A conditional branch for branch target to BLT & a fall-through edge

while decreasing flexibility for relocating basic blocks that terminate with one. In this technique, branches that target the next instruction are removed and execution *falls-through* into the next basic block. However, once we relocate the source or the target basic block, we have to insert a branch instruction at the end of the source basic block so that execution still follows the correct execution path. This technique is called *BLT with FT* where *FT* is an abbreviation for *Fall-through*, and offers the best performance among these three techniques.

Table 3.1 summarizes the actions taken during the generation of relocatable basic blocks. In this table, we list the modifications to branch and call instructions as well as to fall-through edges while using any of the three techniques we described earlier. Our system treats call instructions as fall-through edges. Conditional branches require special care as they have 2 targets. After our transformations, generated basic blocks do not strictly obey the 'basic block' definition for 'BLT with TAT' and

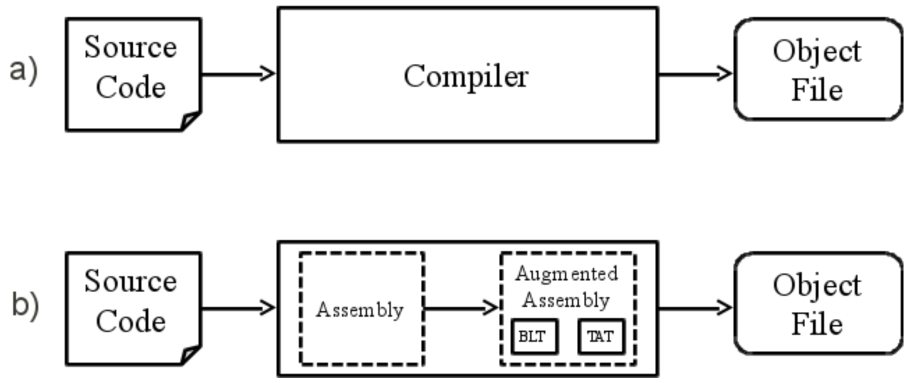


Figure 3.2: Compilation mechanism: a) Regular compiler. b) Our compiler: It generates an intermediate assembly file and augments it with tables.

‘BLT Only’; however, for simplicity, we still refer to them as ‘basic blocks’.

3.3 Compilation

In Sect. 3.2, we discussed the types of relocatable basic blocks. In this section, we explain how they are created. An overview is given in Figure 3.2.

In order to create relocatable basic blocks, we first compile source code into a position-independent intermediate representation using the *-fPIC* flag. This intermediate representation allows us to operate on basic blocks and generate branch instructions to/from a BLT as described in Table 3.1. We selected assembly code as our intermediate representation to have more control on the final executable, but we could have used any other intermediate representation as well. At this stage, we only use labels to represent basic blocks as their final addresses are not known in this intermediate representation. Then, this modified intermediate representation is compiled into an object file just like any other assembly code. Object files are then in turn linked to create executables.

To allow seamless generation of relocatable basic blocks, we developed a wrapper for *gcc* and *g++*. This wrapper acts like a regular C/C++ compiler and supports all flags supported by *gcc* and *g++*. We also created a simple parser for the assembly code. This parser identifies boundaries of basic blocks and modifies the assembly code generated during the first phase of the compilation. In the assembly code, most basic blocks are already labeled as they are usually targets of branch instructions. However, there are often basic blocks that are only reachable through a fall-through edge. As fall-through edges do not require any branching, these basic blocks are not labeled. Our parser identifies these basic blocks and appends labels to them so that they can be used as targets from BLT.

3.4 Instrumentation

Code modification, or code patching, is the act of updating program code outside of the compile-link-load-execute cycle. It can be performed at various steps of the compilation process like pre-compile-time using source-to-source translators, or compile and link-time through compiler passes. For this work, we are only interested in post-link-time code modification (i.e. launch-time and runtime). In post-link-time code modification, program code is modified only after the executable is fully compiled and linked. Reasons for postponing code modification until this phase range from unavailability of source code for legacy and third party executables to the need to observe the effects of compiler optimizations.

Code layout has been traditionally determined by compilers which arrange the

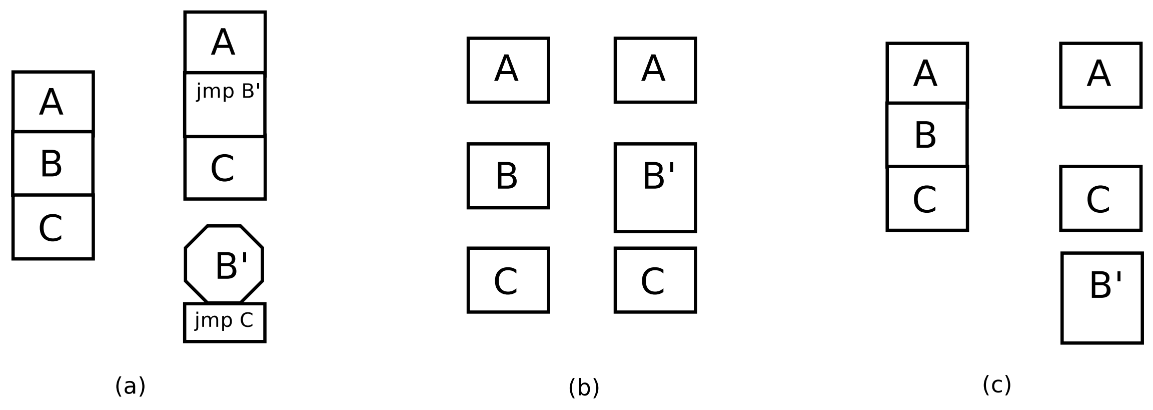


Figure 3.3: Runtime Patching of Basic Block B with regular basic blocks and Relocatable Basic Blocks. (a) Patching a regular basic block normally requires moving it and modifying any address references. (b) With relocatable basic blocks, patching is done in place. (c) If there is not enough room, the target of the patch is moved with very little change in the original code.

layout of individual object files, and linkers which arrange the layout of fully-linked executable files. After linking, all functions and global data have a fixed location within the executable. There usually is not enough room to move any functions or data in this packed layout. As a result, whenever code modification is needed, all basic blocks that are affected are usually moved to the end of the program text and patched there, as seen in Figure 3.3(a). Unconditional branches targeting these basic blocks are inserted at the original locations of these basic blocks. With this approach, the execution can continue after a small disruption. This is the common way tools such as Dyninst [11] perform runtime code modification.

With our approach, these adjustments are mostly unnecessary. Since basic blocks are relocatable, code patching operations do not have to deal with moving the basic blocks and adding branching logic to jump to these blocks. The patch is directly applied as shown in Figure 3.3(b). If basic blocks have to be moved to make space for modifications as in Figure 3.3(c), they are simply moved to new locations and associated BLT or TAT entry is updated to point to their new locations.

The first step to achieve runtime code modification in our system is the same as the first step of traditional instrumentation systems: Users identify the location where instrumentation code will be inserted. Then, the space requirement for the instrumented code is computed by our system. If the target basic block does not have room to grow, it is moved to a new location and associated BLT entry is updated. This step is fairly simple due to the relocatability property of basic blocks. Finally, Instrumentation code is inserted in the target basic block.

Instrumentation is a broad topic, and dealing with some of the required components of instrumentation such as parsing binary files and generating instrumentation code is beyond the scope of the work presented in this chapter. Therefore, we made use of Dyninst's already available features to analyze binary code and generate instrumentation.

3.5 Benefits

In this work, we developed a simpler instrumentation mechanism than a common instrumentation tool employs. Our approach is based on Dyninst which is already a fully-functional tool; therefore, we put our effort into simplifying this tool, rather than adding new functionality.

During a traditional instrumentation, a basic block is relocated to new memory, and a branch instruction is inserted in the old location. If there are multiple modifications at a given point, snippets of instrumentation are inserted into the program image, and they are linked with branch instructions, forming a chain of instrumentation snippets. Our system does not use this logic as it can easily relocate basic blocks. Moreover, our tool can generate inlined instrumentation[1]. Inserting a branch to the original location of an instrumented basic block can be problematic when the relocated basic block is smaller than the required branch instruction. Traditional instrumentation tools set up a trap[2] to redirect execution in these cases. Since relocatable basic blocks do not need these branches, traps are also not necessary. Therefore, relocatable basic blocks can be instrumented without complicated trap logic.

Relocatable basic blocks can also decrease the number of branches if the binary is instrumented heavily. Traditionally, if a basic block B is instrumented, execution has to go through the original location of B to jump to the instrumented block. With relocatable basic blocks, this jump might be unnecessary if the source basic block already looks up the address of the target basic block from the BLT. If most blocks are instrumented, the savings might be considerable.

3.6 Experimental Results

In this section, we present our findings about the overhead caused by relocatable basic blocks and how running times for instrumented binaries compare.

[1]Current version of Dyninst also inlines instrumentation code and performs full function relocation for instrumented functions. We observed the improvement described here over an earlier version of Dyninst that was the most recent release when this work was performed.

[2]A *trap* is set by using a specific instruction. Executing that instruction raises an operating system signal.

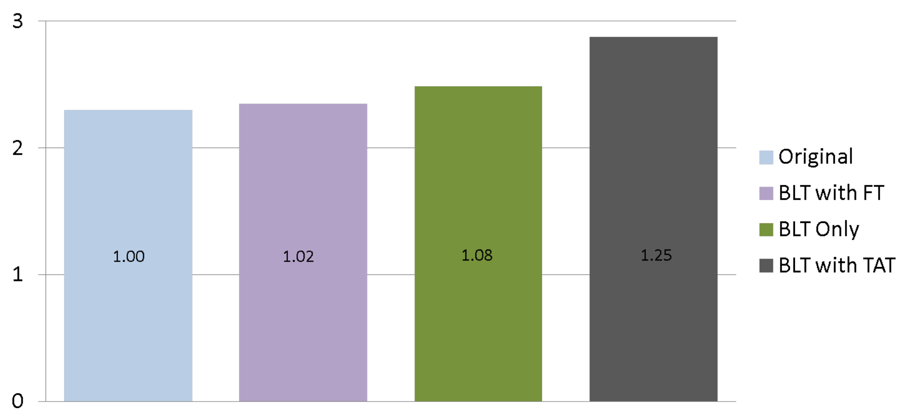


Figure 3.4: Running Time Comparison of Relocatable Basic Block Techniques on Quicksort. y-axis shows the amount of time each code version takes to finish in seconds.

First, we measured the overhead caused by relocatable basic blocks on four versions of an implementation of the quicksort algorithm. To generate these four executables, we first converted the source code into assembly language with gcc using *-fPIC* flag. We modified the resulting assembly code with respect to the strategy we used. We executed each version of the program, including the original. We took an average of the running times across five runs of each version.

Figure 3.4 shows the running times for the original and three alternative code generation techniques. The *y axis* shows the actual running times of each application while the number inside each bar shows the normalized running time for that version of the program. *BLT with TAT* performs the worst with a 25% slowdown. Programs with *BLT with FT* are almost as fast as unmodified programs (2% slowdown was inevitable since our example program included some loops that required taking branches using BLT), while the code is relocatable at the granularity of a small number of basic blocks. However, in this approach, basic blocks that end with fall-through edges have to be appended with branches to replace these fall-through edges

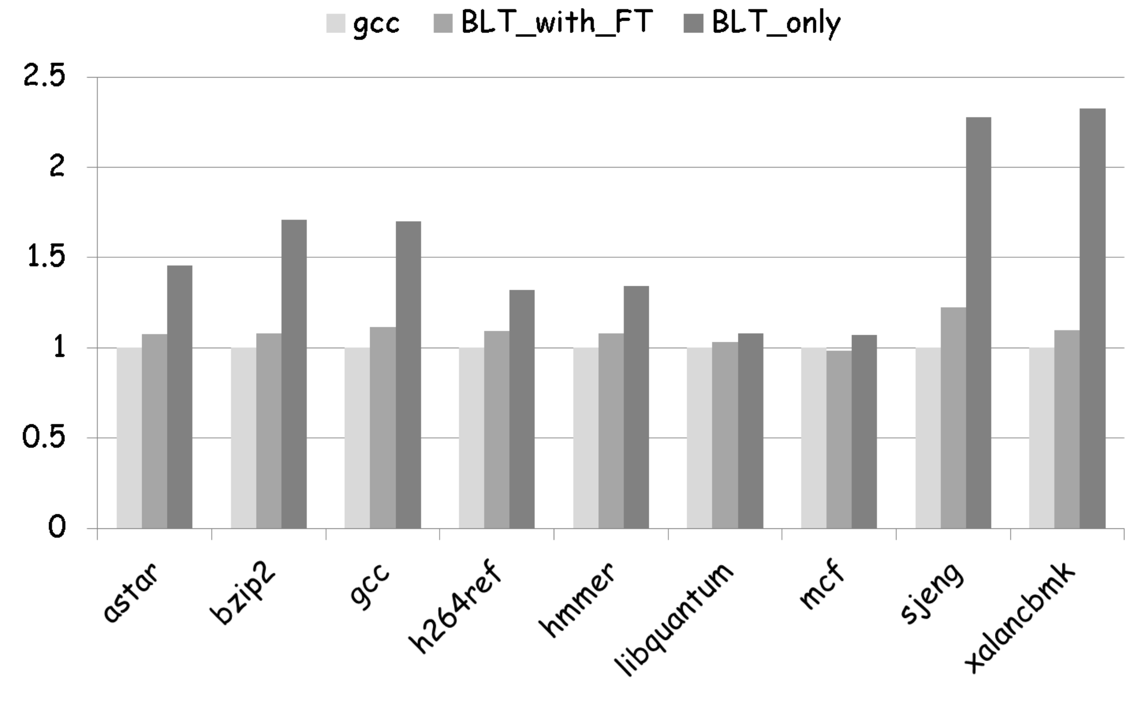


Figure 3.5: Normalized Running Times with Relocatable Basic Blocks

when they are relocated. We claim this slight restriction on flexibility is worth the substantially reduced runtime overhead. Fully flexible code provided with *BLT with TAT* is not as practical as other options due to the high cost on the performance. Therefore, we stopped considering it as a practical option at this point, and we have not run other tests using this strategy.

The next step in our evaluation was to measure the running times of the SPEC CINT 2006 benchmarks compiled with and without relocatable basic blocks. Figure 3.5 shows normalized running times of these benchmarks on the reference data set. Our results showed that *BLT with FT* performs about 9% slower on average than the original executables built with gcc. *BLT only*, on the other hand, performs almost 60% slower on average. These results also show that the most practical strategy to create relocatable basic blocks is to allow fall-through edges between basic blocks.

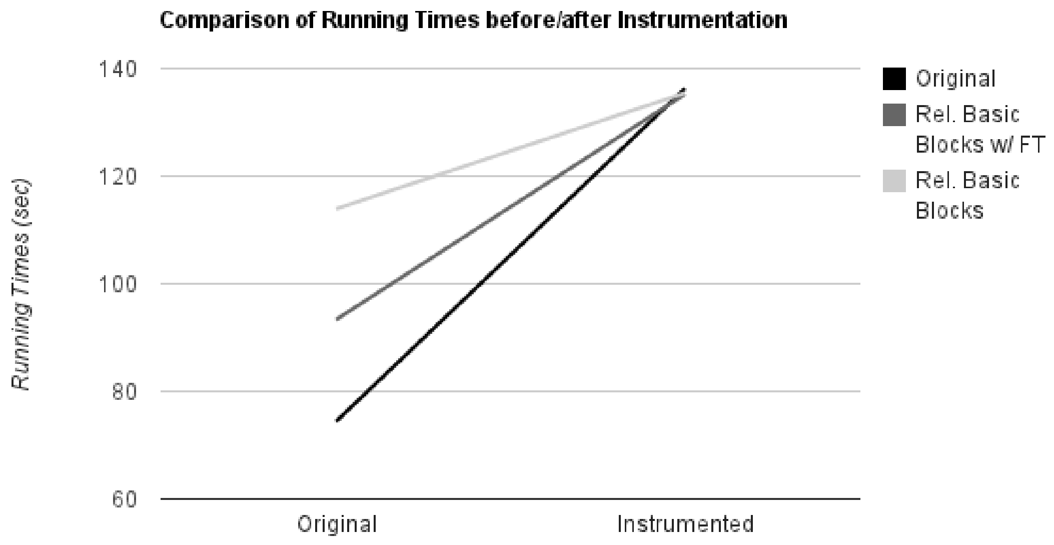


Figure 3.6: Running Times of Instrumented Binaries with Relocatable Basic Blocks

In our next experiment, we compared the running times of binaries instrumented with original Dyninst and our version of Dyninst that is capable of making use of relocatable basic blocks. During instrumentation, Dyninst generates copies of functions that contain both instrumentation code and original code. When a function is instrumented again, the instrumentation is applied to the original function and not to the copies. Therefore, only basic blocks in the original function need to be relocatable. Our instrumentation tool inlines instrumentation code and relocates functions as a whole. Figure 3.6 shows the running times of original and instrumented *mcf* executables. We used three techniques to create these executables: We compiled the program with *gcc*, with *BLT with FT*, and with *BLT only*. *mcf* with *BLT only* takes more time to finish than the one with *BLT with FT*, and even more time than the one with *gcc*; however, after the instrumentation, all three versions

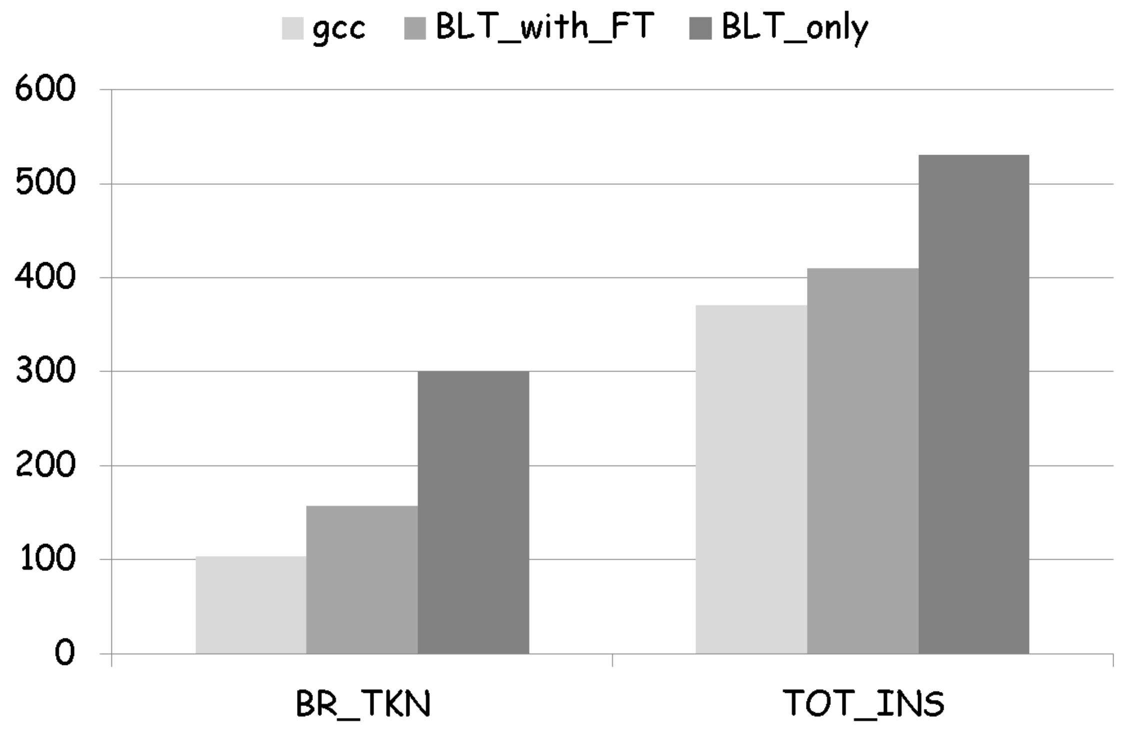


Figure 3.7: Instruction Counts with Relocatable Basic Blocks

of the instrumented *mcf* executable run at about the same speed. These results show us that executables built with one of our techniques do not perform worse after the instrumentation, and we should consider building executables with one of our techniques if we expect to instrument them, as instrumentation of executables built with our technique is simpler than instrumentation of executables built with a regular compiler.

Inevitably, executables built with one of our compilation techniques perform worse than executables built with a regular compiler when there is no instrumentation. We wanted to make sure that this slowdown comes from the fact that we execute more branches for executables built with our techniques. Therefore, we measured the number of executed instructions and number of taken branches on the *mcf* benchmark using the hardware counters available on the system. Figure 3.6 shows that the increase in the number of executed instructions matches the increase in the number of taken branches, suggesting that the slowdown is merely a result of executing these extra branch instructions.

3.7 Discussion

This work demonstrated the benefits of using relocatable basic blocks and the overheads associated with using them. Instrumentation with relocatable basic blocks is simpler, and tools for instrumenting these binaries are easier to build. Moreover, instrumented executables with our mechanism run as fast as instrumented executables with original Dyninst that is more complex than our system.

However, relocatable basic blocks can be impractical as the benefits may not outweigh the drawbacks in all situations. In our work, we arrived at the conclusion that tool builders prefer creating tools with better performance even though building these tools takes more time due to their complexity. As a result, we did not take relocatable basic blocks idea further and applied our ideas about relocatable code to other program constructs in our later work as described in Chapters 5 and 6.

3.8 Conclusion

In this chapter, we introduced a novel mechanism to create individually relocatable basic blocks. Relocatable basic blocks provide more flexibility than shared libraries that have to occupy a contiguous space in memory. Our relocatable basic blocks can move around in the memory. This flexibility provides an easier runtime instrumentation opportunity for tool builders.

Executables with relocatable basic blocks run slower than regular executables due to the increased number of branches that provide relocatability. When instrumented, both executables with relocatable basic blocks and regular executables run at about the same speed. Therefore, we believe executables should be built with relocatable basic blocks if users anticipate to instrument them.

Chapter 4

Compiler Help for Binary Analysis Tools

In Chapter 3 we discussed one approach to simplify instrumentation via relocatable basic blocks. In this chapter, we introduce another approach towards the same goal: a compilation mechanism that speeds up binary parsing operations. We describe the difficulties of binary parsing, followed by our approach and how it simplifies the creation of Control Flow Graphs. Finally, we present our experimental results.

4.1 Overview

Binary analysis is a common operation for performance modeling [51, 63], computer security [27, 74], maintenance [14, 40], and binary modification [19, 69, 80]. Each of these tasks requires parsing the executable file to identify functions, data segments, and their interaction with each other. However, parsing executables is not a straightforward task and it is painfully slow since it usually requires decoding every single instruction in the binary.

At the higher level, even distinguishing code and data is difficult since they are often stored in adjacent memory. It is usually hard to draw the conclusion that a sequence of code constitutes a function, especially when it is not accessed via a statically identifiable call or branch instruction. Therefore, even seemingly-easy tasks such as locating functions might be daunting if the file is stripped from its symbols. At the lower level, identifying instructions in a binary is not a simple task either, especially for the variable-length instruction set architectures such as the x86. When the start of an instruction is not known for certain, the problem of distinguishing data and code is even harder to solve. Various analyses use different methods and almost all of them make use of the available symbols included in the executable file.

All the information about functions and data locations is actually known during various stages of compilation. Compilers decide where each of the components in a binary should be located. They know what is data and what is program code. They know where each function is placed, and they know the boundaries of the instructions. They also have the complete information about the locations of basic blocks and edges between them. They use all this information to build executable files, and then throw away most of the information used during this process. Only some information is stored in the binary in the form of symbols. Binary analysis tools that operate on these executables have to regenerate the information that is thrown away by the compiler.

In this chapter, we propose a novel compilation mechanism that stores useful information about the layout of executable files in tables inside executable files. These tables enable identification of basic blocks and provide support for reconstruction of edges between them. Binary analysis tools that make use of these tables can parse executables faster and more reliably. We measured a speed-up in parsing up to 4.4x with an average speed-up of 3.8x. Since these tables are stored in a section that is not loaded into the memory during execution, the memory footprint of executables do not change. Running times of these executables also remain unchanged since we do not in any way modify the execution. The overhead in the compilation time and the increase in file size is manageable - both values measured to be 23%. At the moment, this compilation mechanism and related binary analysis tools are supported on the x86_64 architecture and with the gnu compiler suite.

4.2 Difficulties of Binary Parsing

Parsing is not trivial. To motivate our compiler based approach, we briefly review some of the parsing related challenges.

The first challenge in parsing the machine code is distinguishing code from data. Since both code and data are stored the same way, there is really no easy way of identifying whether a sequence of bytes correspond to code or data. Current parsing techniques use hints to identify code and mark the remaining bytes as data. These hints usually come in the form of symbols in the binary. Symbols that represent functions indicate where those functions start. Binary analysis tools mark the bytes addressed by these symbols as function entry points. From this point on, tools either follow a sweeping or a recursive strategy [71]. In the sweeping strategy, tools first use symbols to mark an initial set of functions, then sweep the remaining bytes from the start of the file and mark sequences of bytes that resemble code as program code. In the recursive strategy, tools also start by using symbols to mark the initial set of functions. Starting from these functions, tools then locate other code sections following call edges[1] and marking call targets as function entry

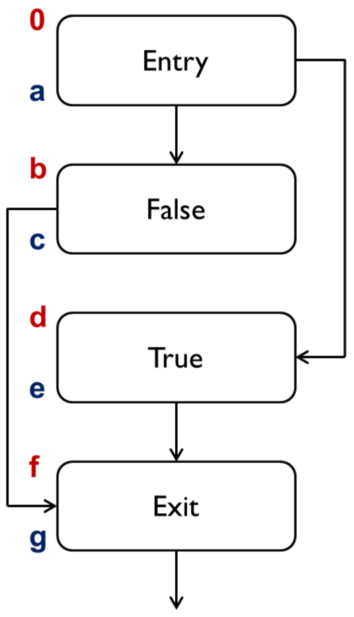
[1]Some function invocations are performed using branch instructions. The sequence of bytes

points, hence program code. In some cases, uncharted regions in the binary are then plugged into machine learning algorithms to identify even more functions and code regions [70].

Functions are composed of one or more, usually several, *basic blocks.* A basic block is a sequence of instructions that contains no control flow instructions except as the last instruction of the block. If the first instruction in a basic block executes, it is guaranteed that all following instructions will execute. It is an abstraction that is used by many types of analyses.

Once the functions are identified, their *Control Flow Graphs* (CFGs) are built. A CFG is a graph whose nodes are basic blocks and whose edges represent the interactions between basic blocks. It can be considered as a reachability graph starting from the function entry point. A sample CFG can be seen in Figure 4.1. The function in this sample contains an entry block, two blocks that represent an If/Else (labeled True and False) structure, and an exit block. Building such a CFG correctly depends on correct identification of basic blocks and the edges between these basic blocks. Therefore, it requires the analysis tool to inspect each instruction in a function. This operation is error-prone, especially on variable-length instruction set architectures where an error in decoding an instruction propagates downstream and make decoding the following instructions harder, or even impossible. Since each instruction has to be decoded and analyzed, building a CFG is very costly. This cost matters because building a CFG is the first step for most binary analysis algorithms.

addressed by branch instructions is also considered code.



First Instruction	Last Instruction
0	a
b	c
d	e
f	g

Basic Block Table

Source	Target	Type
0	b	Fall-through
0	d	Conditional
b	f	Unconditional
d	f	Fall-through
f	N/A	Return

Edge Table

Figure 4.1: A sample CFG and associated Basic Block and Edge Tables of a function with entry basic block, an If/Else structure, a loop, and an exit block.

4.3 Compiler Help

During the build process, compilers construct an internal representation of the source code and generate machine code using this internal representation. However, as soon as the executable file is generated, this internal representation is thrown away. In this chapter, we investigate the effects of storing some of this knowledge about the program inside the generated executable.

In the previous section, we explained that building CFGs is a precursor to most binary analysis algorithms and that building a CFG requires identifying basic blocks and edges between them. Most of the time spent while parsing an executable file is used to gather this information. We demonstrate how storing this information inside the executable file speeds up the parsing process. Moreover, we show that the extra information stored in the executable file does not change the speed of a)

<First Instruction: Offset>	<Last Instruction: Offset>

b)

<Source: Offset>	<Target: Offset>	<Type: Integer>

Figure 4.2: Formats of Basic Block and Edge Tables: The first line shows the format of the Basic Block Table whereas the second line shows the format of the Edge Table.

execution of the program or the size of the memory image of the program. The following subsections explain what extra information is stored and how it is used.

4.3.1 Basic Block and Edge Tables

To speed up building basic block abstractions inside binary analysis tools, we developed a compilation framework that stores the start address[2] and the address of the last instruction of each basic block inside the executable in what we call a *Basic Block Table*. The format of this table is shown in Figure 4.2 a). To create a basic block abstraction, tools can read the first field and find out where that basic block is located. The second field is used for gathering information about edges. A more detailed discussion about this field will follow after we introduce the *Edge Table*.

CFGs also require the identification of edges between basic blocks. Our system stores the source basic block, the target basic block, and the edge type of each edge in the *Edge Table*. In this table, a basic block is identified by its start address. The format of this table is shown in Figure 4.2 b). In some cases, target basic block in an edge can only be known during the actual execution of the program. Such cases occur when value of a function pointer or any sort of indirect branch target[3]

[2]The values used as address are offsets from the start of the function. We leave the discussion on using offsets rather than absolute addresses to Section 4.3.2

[3]Although indexed jump tables also use indirect branches, targets of such indirect branch instructions can usually be identified using heuristics. Targets cannot be identified if heuristics do

depends on the program inputs. In these cases, we leave the target basic block field blank during generation of the Basic Block Table. Such edges can be filled in by the parser after the program has launched and when the value of the target address can be computed through the analysis of the last instruction of the source basic block. The last instruction of a basic block is also accessed when binary analysis tools need to modify the CFG. Updating the last instruction in a basic block to change the target of that basic block modifies the edge, and thus is a common CFG update operation. Clearly, we need a quick way to access the last instruction of each basic block. Therefore, we store the address of the last instruction in our Basic Block Table along with the start address of that basic block.

Basic Block and Edge Tables shown in Figure 4.1 are based on the CFG from the same figure. For this example, assume a is the offset of the last instruction of the first basic block from the start of the function, c is the offset of the last instruction of the second basic block, and so on. Since there are 4 basic blocks in the CFG, the Basic Block table has 4 rows. The first row represents the first basic block: it starts at offset 0, and its last instruction is located at offset a. The second basic block starts at offset b, and its last instruction is located at offset c, and so on. The Basic Block Table is followed by the Edge Table which has 5 rows, one for each edge between basic blocks. The edges from the entry basic block to the *True* and *False* blocks are shown in the first two rows. The edge from the entry block to the *True* block is represented with the triplet of <*0, d, Conditional*> since it is accessed by taking a conditional branch. The edge from the entry block to the *False*

not work, or if indirect branch instructions do not use an indexed jump table.

block is traversed when the conditional branch is not taken and when execution simply falls-through[4] to the *False* block. The last row in the figure represents the edge originating from the return instruction. Since the target of a return instruction cannot be determined statically, that field is left blank (marked with N/A in our example).

4.3.2 Compilation Process

Our compilation mechanism mimics a standard compilation process from the end-user's point of view as much as possible. We developed a tool that modifies assembly files and generates Basic Block and Edge Tables. This approach was chosen because no compilers we are aware of are able to generate the tables that we need. We considered modifying an open-source compiler and generating the tables directly using this compiler; however, this approach would require substantial development time. Therefore, we decided to defer this task and instead modify the intermediate binary files. Using an LLVM [53] pass was another approach we considered but did not take to avoid unintentional optimizations LLVM applies to binaries with its standard passes.

Figure 4.3 shows an overview of our compilation process: a) shows regular compilation where source code is converted into an object file directly by the compiler. Our compilation process is shown in b). We first convert source code into an assembly file. Our assembly modification tool then generates Basic Block and Edge

[4]'Fall-through' is a type of control transfer where the execution of an instruction is followed by the execution of the next instruction in the address space of the executable.

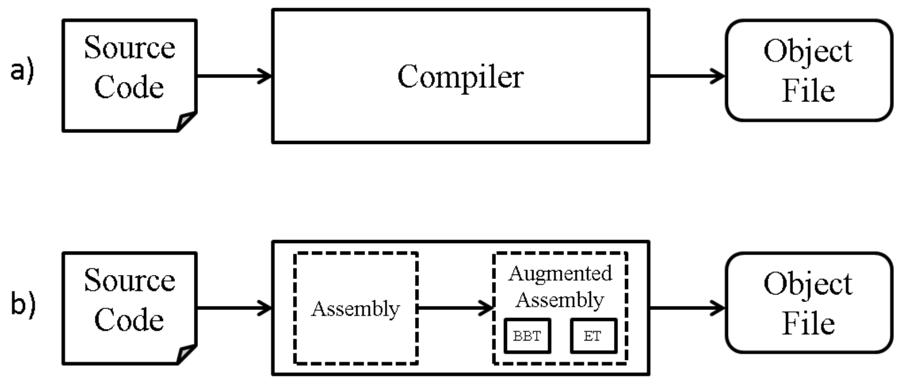


Figure 4.3: Compilation Mechanism for Basic Block and Edge Table Creation: a) Regular compiler. b) Our compiler: It generates an intermediate assembly file and augments it with tables: Basic Block Table (BBT) and Edge Table (ET).

Tables. The assembly code is then rewritten to disk and it is appended with one Basic Block Table and one Edge Table per function. For our purposes, the critical information about programs is still available at the assembly level. The formats of these tables were discussed in Section 4.3.1. This augmented assembly file is finally converted to an object file.

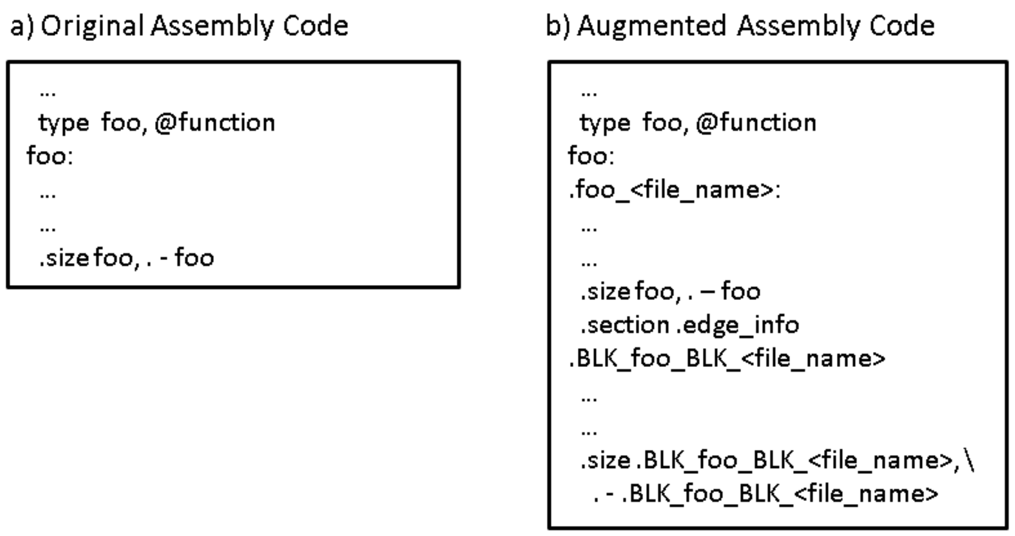
In order to generate Basic Block and Edge Tables, our assembly modification tool has to go over the assembly code and identify functions. Figure 4.4 shows original and augmented assembly files. In the context of assembly files, functions are regions of code in the *.text* section and are identified by *@function* markers. At the end of the function block, a *.size* directive is used to calculate the size of the function. Each function contains one or more, usually several, basic blocks. Basic blocks either start at a label (when it is the target of some branch instruction), or right after a call or a branch instruction as the fall-through target. Once the boundaries of basic blocks are identified, our tool also marks the last instruction in each basic block.

Next, our assembly modification tool identifies the edges between basic blocks. It reads the last instruction of each basic block, determines the type of the edge (e.g. *call*, *return*, *(un)conditional branch*), and finds out the target basic block if possible. There may be more than one target for some instructions such as conditional branch instructions which have a branch target and a fall-through target.

Once the basic blocks and edges are identified, our tool rewrites the assembly file as shown in Figure 4.4 b). Our parser adds Basic Block and Edge Tables to the end of the function in a section we call *.edge_info*. We mark this section with a flag to indicate it should not be loaded into memory during execution.

One minor trick in the implementation is the need to support position independent code as well as position dependent code. Using absolute addressing does not work for position independent code. To handle this, all the addresses in these tables are stored as offsets from the start of the function.

Another issue arises when duplicate function definitions are merged by the linker. During the build process files that are linked with the *include* directive from a source file are compiled along with the actual source file. As a result all function definitions included from a header file are compiled into the resulting object file. If the same functions are linked from multiple source files, these functions will appear in multiple object files that result from the compilation of these source files. To avoid linking problems, these functions are marked as *weak*. Linkers allow only one copy of these weak functions to appear in the final executable - the remaining ones are dropped. Although these functions are identical at the source code level, since compilers perform optimizations individually on each copy of the function, the


a) Original Assembly Code

```
 ...
 type foo, @function
foo:
 ...
 ...
 .size foo, . - foo
```

b) Augmented Assembly Code

```
 ...
 type foo, @function
foo:
.foo_<file_name>:
 ...
 ...
 .size foo, . – foo
 .section .edge_info
.BLK_foo_BLK_<file_name>
 ...
 ...
 .size .BLK_foo_BLK_<file_name>,\
  . - .BLK_foo_BLK_<file_name>
```

Figure 4.4: Regular vs. Augmented Assembly Files: a) A function is shown in assembly format. Anything between *@function* marker and *.size* directive is considered part of the function. b) The function is augmented with Basic Block and Edge Tables, and a shadow symbol to store the file name.

resulting machine code may be structurally different. As a result, Basic Block and Edge Tables for these functions may differ slightly.

Unlike weak functions, tables with the same name cannot be merged. Therefore, our compilation mechanism has to distinguish tables related to functions that have the same name. Therefore, we generate table names using a combination of the function name and the name of the file that contains that function. At the end, each table has a different name. We also store a shadow symbol inside each function that shows the name of the file where this function is defined[5].

When weak functions are merged, only one such symbol survives. We use that symbol to identify the name of the file that contains the weak function that made its way into the final executable. Binary analysis tools can find the corresponding

[5]The shadow symbol for function *foo* in Figure 4.4 is *foo_ <file_name>* where *<file_name>* is the actual file name of the source.

tables using the file name and function name tuple.

When this rewrite operation of the assembly code is complete, it is assembled into the requested output format such as an executable file or an object file, or simply left as an assembly file.

Our tool supports standard gcc flags. Flags that are not used by our compilation mechanism are passed to the underlying system-supported compiler. The flags that determine the requested output format such as *-c*, *-o*, and *-S* are handled specially by our tool and are not passed to the system compiler to avoid creating output files prematurely. We also provide wrappers for standard compilers in gnu compiler suite. These wrappers, along with the fact that we pass user-specified flags directly to the underlying compiler, provide a seamless end-user experience.

4.3.3 Reconstruction

Parsing binaries is considerably easier when Basic Block and Edge Tables are available. Any binary analysis tool first has to find the location of each table. At that point, the name of the function that is being parsed is known by the tool. Using the function name, the tool searches for the shadow symbol and extracts the file name from this shadow symbol once it is found. The tables are then located using the combination of the function name and the file name. Once the tables are found, the tool reads in the information regarding the location of each basic block and its last instruction. At the end of this step the basic blocks are created, and the locations of their last instructions are known.

The tool then reads in the information about the edges. Each line in this

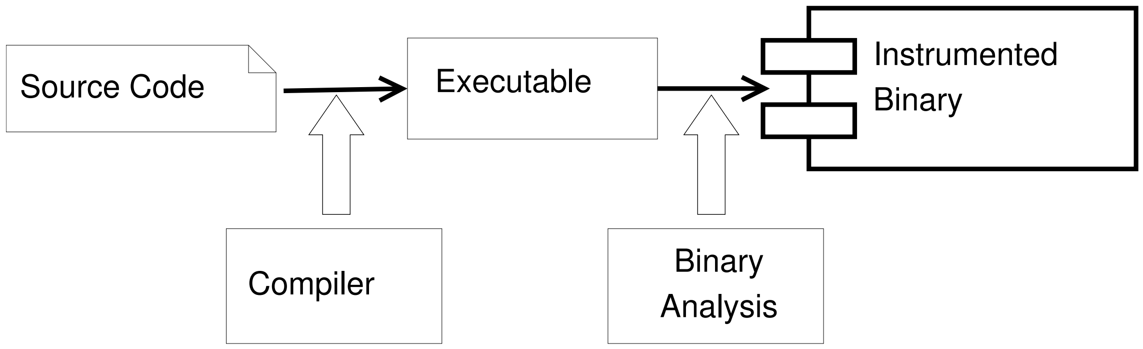


Figure 4.5: Interaction between Compilers and Binary Analysis Tools: Our compiler creates executable files with Basic Block and Edge Tables. Binary analysis tools parse these executables using these tables.

table contains a triplet: the address of the source basic block, the address of the target basic block, and the type of the edge. Since the start address of a basic block uniquely identifies that basic block, this triplet has all the information the tool needs to build the edge abstraction.

As we mentioned earlier, all addresses in these tables are stored as offsets from the start address of the function in hand. To calculate the exact location of a basic block, our tool adds the start address of this function to the value read from these tables.

Figure 4.5 demonstrates the relationship between our compilation infrastructure and binary analysis tools: Our compiler converts the source code into executable files whereas binary analysis tools operate on executable files directly. Binary analysis tools do not interact with our compiler. Therefore, any binary analysis tool can implement the functionality to read the tables stored in these executable files. We built our binary analysis tool based on Dyninst [11] since it is a popular open-source binary analysis library. We modified the ParseAPI component and only rewrote a few functions to allow Dyninst to use the data from our tool.

4.4 Evaluation

For evaluation, we used a simple binary modification tool we built on top of Dyninst. Our tool reads in a binary file and rewrites it to disk with simple instrumentation code for basic block counting. Since our analysis deals with every single basic block in the executable, the executable file has to be parsed from top to bottom correctly locating every basic block.

We evaluated our system on a variety of benchmarks to determine how well our system handles executables with different properties. We first evaluated our system on SPEC CINT2006 [38, 61]. SPEC CINT2006 contains a series of CPU-intensive executables that are selected to evaluate the processor(s) and the memory system. All together, SPEC CINT2006 has about 1,047,000 lines of code.

Our next benchmarks were the PETSc libraries and their sample applications [4]. PETSc (Portable, Extensible Toolkit for Scientific Computation) "is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations." It uses MPI for parallelization. It has linear and non-linear equation solvers and supports C, C++, Fortran and Python. The PETSc suite is composed of about 872,000 lines of code.

Finally we evaluated our system on the popular web browser Firefox (version 9.0.1) [64] and all the shared libraries that ship with the Firefox source code. We evaluated our system on Firefox because its executables are numerous and are relatively large. Moreover, it contains hand-written assembly files and the build process involves using many uncommon compiler options. Therefore, building Firefox has been a valuable test for the robustness of our compilation mechanism. The Firefox suite contains approximately 5,335,000 lines of code.

4.4.1 Environment

All experiments were carried out on 64-bit x86 machines that run Linux operating system.

SPEC CINT2006 and PETSc benchmarks were tested on a system that has 4 Intel Xeon processors with 6-cores each, all clocked at 2.4GHz. It has a total of 48GB main memory. All our executables except PETSc executables were serial applications including our own analysis. Therefore, we ran most of our experiments serially on a single core. Since the purpose of our work is not parallel evaluation, we only used 6 cores for PETSc. Using 6 cores gave us enough parallelization to finish executions in a reasonable time. In our experiments, the extra cores were not used and left to the operating system as a way to reduce operating system jitter. We used gcc 4.1.2 for building reference executables and as a back-end to our compilation mechanism.

Firefox experiments were run on a separate machine due to the idiosyncratic requirements of the Firefox build environment. As a result, Firefox runs were taken on a dual-core machine with an AMD Turion processor at 1.8 GHz with 2GB main memory. Since we never compare results across these machines directly, the results are not affected due to using two different machines. On this system, we used gcc 4.6.1 for building reference executables and as our back-end.

4.4.2 Experimental Results

Our first experiment was designed to calculate the time it takes to parse a specific executable using our analysis tool to show how much our tool improves parsing speed. We then ran other experiments to evaluate properties of executables built using our compilation mechanism and identify any trade-offs. In this regard, we compared the compilation times using our tool versus a standard compiler. Similarly, we compared file sizes after the compilation process. At the end, we tested the runtime performance of these executables in terms of time and memory usage. We ran each experiment that measures elapsed time 5 times and computed the mean. We then normalized our findings with respect to the executables compiled with regular gnu compiler suite.

In our experiments, we observed that using basic block and edge tables reduced the parsing time between 58% and 77%, and on average by 73%. Although the file sizes increase by 23% on average, we believe this situation is not prohibitive since the basic block and edge tables are not loaded into memory during the execution of these binary files. We also observed about 23% increase in the compilation time. Since this is only a one time cost that appears while building executables, and it can be improved drastically by integrating the creation of basic block and edge tables into the compiler rather than leaving as a separate assembly pass, we believe this increase is acceptable.

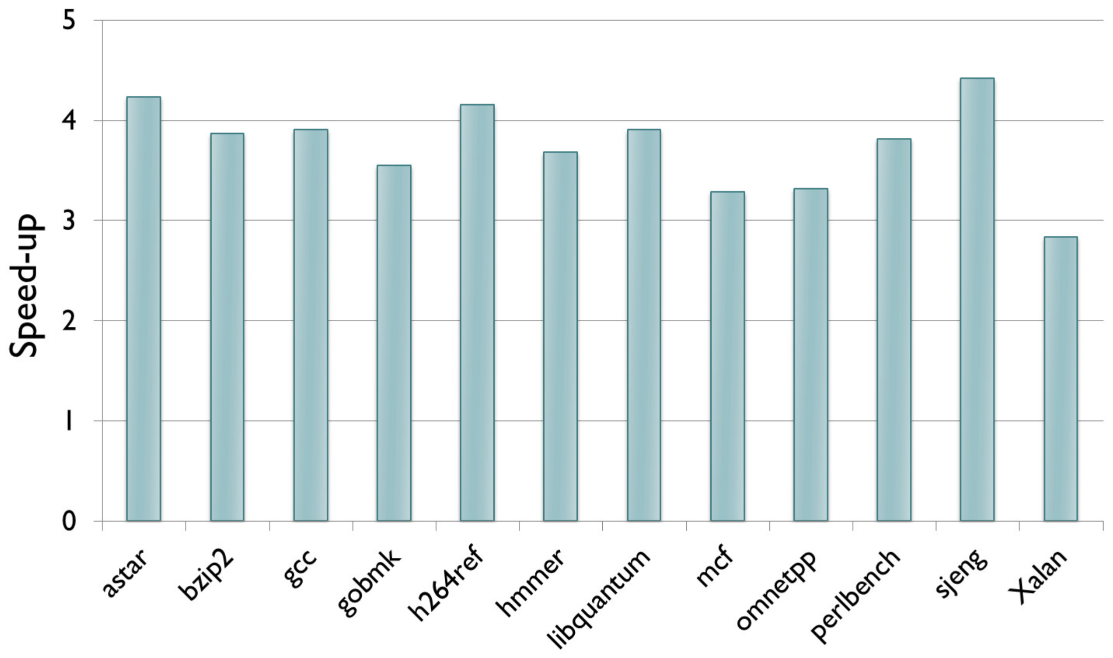


Figure 4.6: SPEC CINT2006 Benchmarks: Normalized Parsing Times

4.4.2.1 Experimental Parsing Results

Figure 4.6 shows the normalized parsing times with SPEC CINT2006 benchmarks with respect to regular parsing[6]. We observed a high percentage of speed-up across the board while the average binary parse speed-up is 3.7x (73% improvement over original parsing time).

Our next tests were carried out on executables in the *snes* package of the PETSc suite and the results are shown in Figure 4.7. One interesting characteristic of PETSc executables is that PETSc libraries are statically linked into the executable files during compilation. As a result, each executable contains all functions in the PETSc libraries. With the help of our compilation mechanism, we reduced the

[6]Note that *gcc* is a benchmark in SPEC CINT2006. We built this benchmark with our compilation mechanism and compared parsing times with the parsing times of the same benchmark (i.e. *gcc*) built with the system supported gcc compiler.

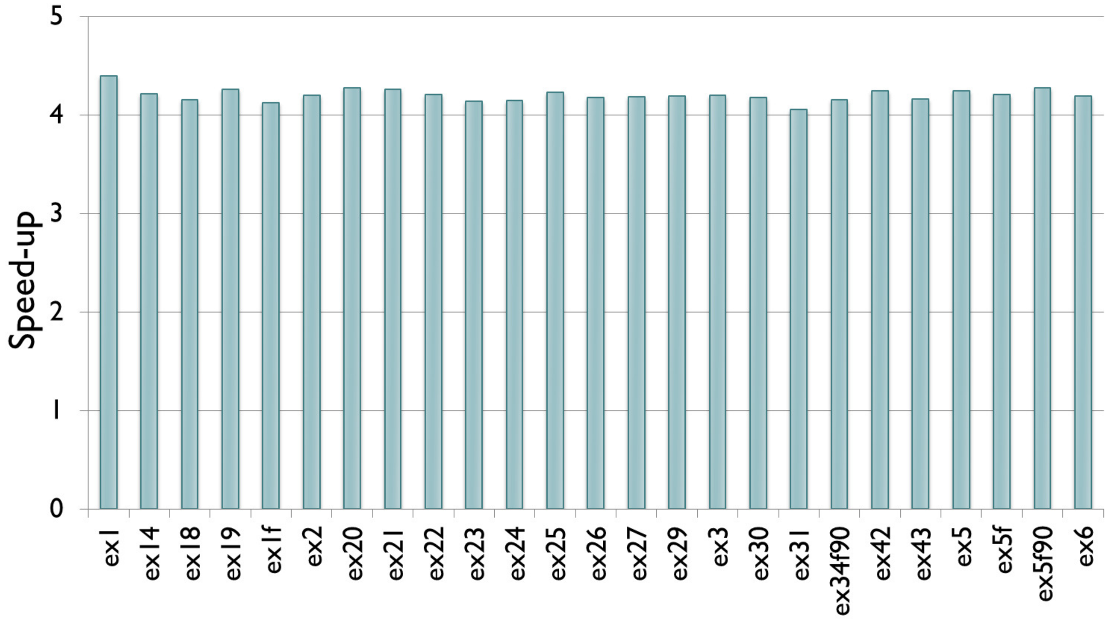


Figure 4.7: PETSc Example Applications: Normalized Parsing Times

parsing time 76% on average (4.2x speed-up). Due to static linking of all these PETSc libraries in every executable, parsing time is more or less flat across all executables in this set because our tool parses mostly the same set of functions for each executable.

As a final set of executables, we decided to use the Firefox executable and all shared libraries that ship with Firefox. Figure 4.8 shows the normalized parsing time of executables from Firefox. For this set of runs, we operated on those executables that reside in memory when the Firefox web browser is launched. We see a major improvement in parsing time once more as expected. The average drop in the parsing time is 71% (3.5x speed-up) with the worst case reduction of 58%.

As the previous results show, our system considerably increases the parsing speed. Now we want to discuss other evaluation metrics such as file size, compilation time, and memory footprint of executables compiled with basic block and edge

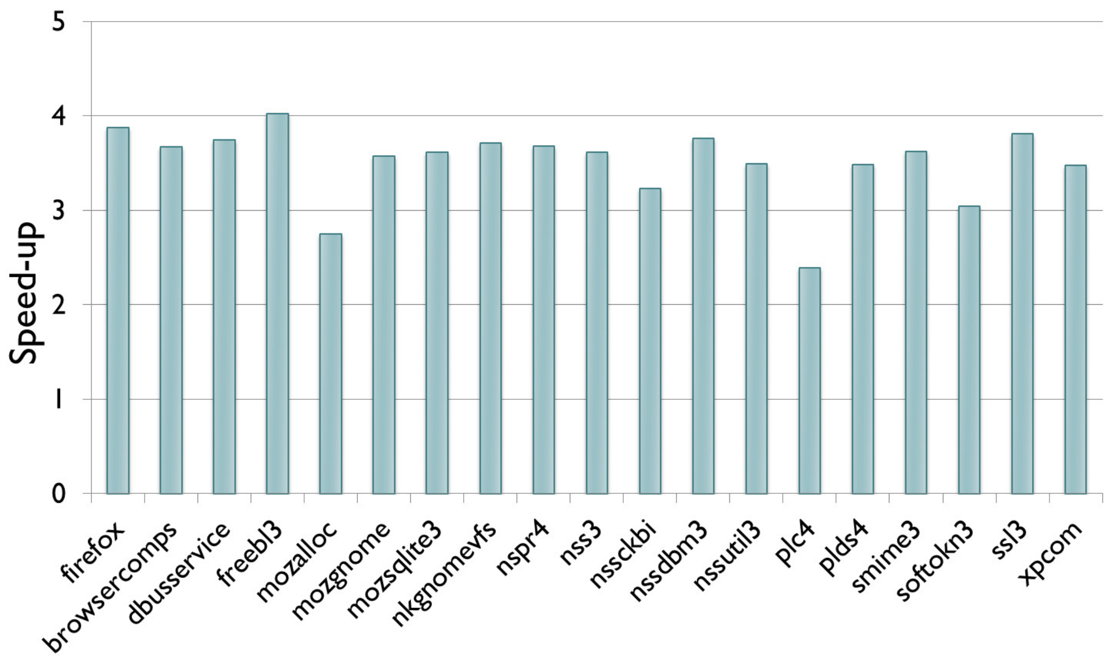


Figure 4.8: Firefox Executables: Normalized Parsing Times

tables.

4.4.2.2 Build Time Metrics

Table 4.1 gives an overview of our experimental results regarding build time and runtime metrics. In this section, we will discuss the build time metrics: file size and compilation time.

Table 4.1: Properties of Executables Built with Basic Block and Edge Tables (All numbers are normalized)

Benchmark Set	File Size		Compilation Time	Running Time	Memory Footprint
	w/o Debug	w/ Debug			
SPEC CINT2006	2.21	1.38	1.25	0.97	1.00
PETSc	1.50	1.09	1.32	0.95	1.00
Firefox	1.17	1.21	1.13	0.94	1.00
Average	1.63	1.23	1.23	0.95	1.00

Since we are adding extra data to executable files, the size on disk unavoidably increases. On average, we are adding about 20 bytes of data to the executable for each basic block, and two extra symbols to the symbol table for each function. Table 4.1 shows the normalized file sizes across three sets of benchmarks along with the overall average. The column *'w/o Debug'* shows the comparison of file sizes when they are built without the debug flag on while the column *'w/ Debug'* shows the comparison with the debug flag (*-g*) on. We show both numbers since we realize executables are often built with debug flag on to improve debugging and other binary analyses on these files. The highest increase compared to the size of the original set of executables, 121%, was observed while compiling the SPEC CINT2006 benchmarks without the debug flag on. The average file size increase compared to the size of the original set of executables was 63% while compiling without the debug flag on, and 23% while compiling with the debug flag on. We assert that this increase is manageable since it does not impact the memory used during execution.

Another evaluation metric we used is the compilation time. Since our compilation mechanism uses an intermediate step to process the assembly code generated by gcc, our compilations take more time than the original compilations. The bulk of the increase in compilation time comes from the cost of processing an assembly file as text, and writing out a modified assembly file, again as text. We understand this step is costly. To be more precise, our experiments showed a 23% increase in the compilation time as seen in Table 4.1. We believe our system can easily be integrated into a mainstream compiler such as gcc. Since compilers already keep track of the information we generate using our own mechanism, the added cost would be minimal: about the same as the cost of writing those tables to the binary. This improvement remains as future work.

4.4.2.3 Running Time Metrics

The next set of metrics we looked at was the running times of the executables built using our compilation mechanism and their memory footprint. The results are presented in Table 4.1. For these experiments, we used workloads provided by the benchmarks. SPEC CINT2006 benchmarks have well-defined workloads: We used *ref* data sets for each benchmark. For PETSc, we used *ex30* executable in *snes* package since it runs long enough for consistent timing (for about 28 minutes on 6 cores on the platform described in Section 4.4.1).

Timing Firefox runs was trickier as it does not ship with any specific workloads. The normalized times we report in Table 4.1 is the time it takes Firefox to finish execution when launched with *--help* flag on the command line. Since this type of execution may hide some of the properties of execution, we also benchmarked Firefox with a JavaScript benchmark, V8 [35]. However, results of this benchmark cannot be converted to running time directly. Therefore, we report these numbers separately in Table 4.2. Readers should note that higher numbers represent better performance in the V8 benchmark.

In our experiments we have not experienced any measurable increase of the execution time of the benchmarks. The slight improvement we observed in the running time after using our compilation mechanism is well within the noise of the experiment. Similarly, V8 benchmark results indicate that performance of Firefox

Table 4.2: Firefox Performance with Tables using V8 JavaScript benchmark (Bigger is better)

Firefox Version	V8 Score
Built with system compiler (gnu compiler suite)	2549.2
Built with our compilation mechanism	2587.6

built with our compilation mechanism is almost identical to the performance of Firefox built with the system compiler.

We expect no memory footprint change from our approach. To verify this, we evaluated each executable under Valgrind's *massif* tool [65]. Standard execution of this tool measures the heap memory used by the target executable. For these experiments, we ran the tool with a flag that also takes stack memory into account (i.e. *--pages-as-heap=yes*). Table 4.1 shows no change in the memory footprint. Results indicate that both stack and heap memory used by the programs remain about the same.

4.5 Discussion

Parsing executable files is the first step for any CFG-based binary analysis. Our experimental results show that our mechanism clearly speeds up parsing executable files. As a result, executables can be analyzed faster. It is not hard to imagine bundling more information with the binary would be useful to speed up other binary analyses, or improve their precision, such as liveness analysis of registers or dependency analysis.

However, there are also shortcomings of our work. One such shortcoming is that our system adds *2n* more symbols into the symbol table where n is the number of functions. Since symbol table formats are highly-optimized for space efficiency and look-up speed, this issue is more like a nuisance than a technical problem.

Another shortcoming is that our tool is not fully reliable for processing hand-written assembly files when branch instructions do not identify targets with labels but rather with offsets from those branch instructions. This sort of assembly code can only be produced by experts through hand-tuning for a specific target architecture. Although this shortcoming can be alleviated with better analysis of assembly code, we observed such cases were extremely uncommon[7] to justify the effort.

Increased compilation times might also be annoying for large frameworks such as Firefox. Although the increase in the compilation time was about 23% in our experiments, this increase translates to approximately 20 minutes with such a long build time. However, we expect that integrating our system with a full compiler would substantially speed up compilation and minimize the extra time compilation takes.

One improvement to our plain text table based system would be compressing Basic Block and Edge Tables to reduce disk space demand. In our preliminary experimentations, we observed that compressing Basic Block and Edge Tables reduced the size of these tables by about 78%. However, since binary analysis tools cannot read compressed tables directly, they would need to decompress them before first use. This approach incurs a runtime performance penalty, and our first aim is to reduce the runtime costs.

[7]In our experiments we only came across 3 such files in the Firefox suite of more than 37,000 source files.

4.6 Conclusion

Parsing binary code is the first step for most binary analyses. However, it is costly and imprecise especially on variable-length instruction set architectures. Still, for situations where source code is not available, there is no chance but to parse binaries.

In this chapter we introduced a novel compilation mechanism that improves the parsing speed of binary files when they are examined by binary analysis tools. Our compiler creates intermediate assembly files, augments them with information about basic blocks and edges between them, and generates executable files using this augmented assembly code.

We implemented an instrumentation program for basic block counting that rewrites a binary to the disk with the instrumentation code using the Dyninst library. We showed that running this analysis code on various benchmarks resulted in up to 4.4x speed-up in parsing time, with an average of 3.8x. Although the size of the binary files increase with extra data in the tables we generate, since these tables are not loaded into memory during execution, the size of the runtime memory image of the executable remains the same as before. Moreover, there is no runtime performance degradation due to these tables.

Chapter 5

Profile-driven Selective Program Loading

Chapters 3 and 4 were about simplifying the instrumentation process. Chapters 5 and 6, on the other hand, are about improving properties of programs using various instrumentation and binary editing techniques.

This chapter introduces a mechanism to reduce the memory footprint of shared libraries by avoiding loading their unused functions. We talk about how shared libraries are modified so that loaders omit some parts while loading them. We then talk about the reduction in the number of memory pages occupied by these shared libraries, and show that the running time of applications do not change. We conclude with a discussion.

5.1 Overview

Software systems have been constantly getting more functional and complex. Most systems are not composed of a single executable file any more; they are a combination of many executables and shared libraries. These executables often use components developed by others. Frequently, users of these general purpose shared libraries are interested in only a fraction of a library's functionality.

Ideally, only the necessary parts of shared libraries should reside in main memory during execution. Traditionally systems have relied on demand paging to only load those parts of libraries that are actually used into memory. However, systems such as IBM's BlueGene and Cray XT series lack local disks on each compute node and therefore avoid virtual memory and demand paging for performance reasons [2, 43]. Thus the available memory on such systems is limited to what is physically available. Moreover, this memory is shared between applications and their data. Reducing the memory footprint of application text space is therefore crucial for large and complex applications that deal with large datasets.

During launch of an application, the executable file and all of shared libraries are loaded into memory [56]. Typical applications today use several large shared libraries and relatively small executable files. A typical PETSc application takes over 16 MB of memory to load the application and shared libraries although the actual executable only occupies 0.02 MB of that space. Many large US DOE applications have text segment sizes of around 100 megabytes.

In this work, we propose a system that reduces the memory footprint of shared libraries by eliminating unused parts of libraries from an executable. Our approach relies on an efficient profiling mechanism that lets us determine a list of functions that are not executed in the common case. We then modify ELF program headers so that these functions are not loaded into memory when the program is launched. If, for some reason, any function that has not been loaded is accessed at runtime, our system includes an error recovery mechanism that loads that function into the memory and allows the application to continue execution.

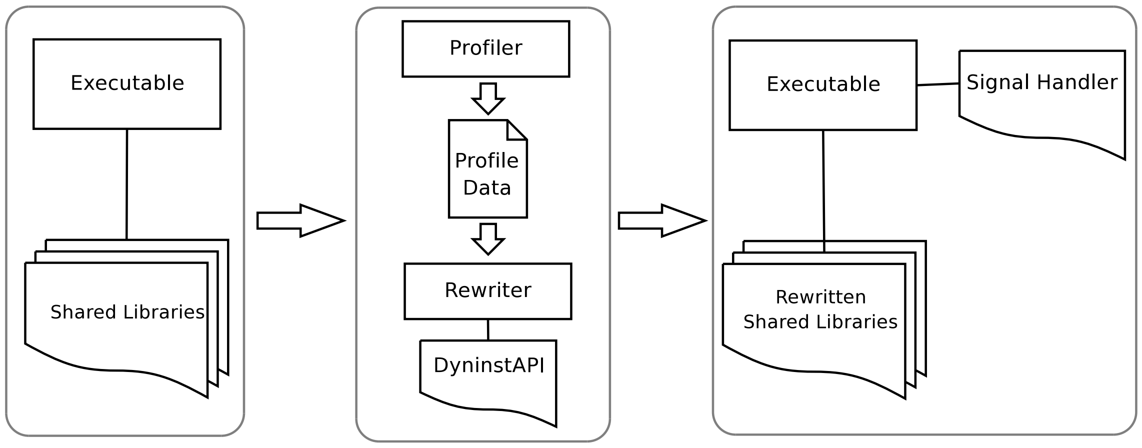


Figure 5.1: Overview of the Selective Program Loading System: Executables and shared libraries are profiled and rewritten

5.2 Architecture

Figure 5.1 shows the architecture of our system. It is composed of a profiler, and a program analyzer / rewriter.

Our design obeys the “make the common case fast” motto. A profiler is used to get a trace of executed functions for a given application. Then, a list of functions that are not used is generated for each shared library. Since code has to be loaded in page-sized units, removing a single function does not save any pages since there is usually other code around that function that still needs to be loaded. Therefore, we need to re-arrange code and group unused functions so that we can remove them from the loadable sections altogether. Our tool moves all unused functions to the end of the code section and modifies ELF program headers to make those parts unloadable. Finally, our tool writes the modified shared libraries to the disk to make the changes permanent.

5.3 Target Applications and Platforms

Our prime target is applications that use a limited number of functions from many shared libraries or frameworks, such as the PETSc. Our system can easily be applied to any shared library on any ELF platform (e.g. most Unix-based systems). For this work, we concentrated our development efforts on one architecture, the x86, since that was the most available cluster at the time. However, high end HPC systems like BlueGene and Cray that do not support demand paging can benefit from our system the most since available memory is scarcer on these systems.

5.4 System Design

Our system is composed of three main components. Profiling is performed to identify a list of functions that are used during training runs of the executable. Profiling data is fed to our analyzer and a binary rewriter which uses Dyninst [11] to access functions, their control flow graphs, basic blocks, and finally instructions. For each shared library, our analyzer and binary rewriter performs the following tasks:

1. Calculation of updated start addresses for each function that is being moved. Functions that are used often will be placed before the functions that are rarely or never used.

2. Code generation for moved functions. Call instructions, address calculations for global offset table (GOT), contents of jump tables, and function pointer calculations are updated using the new locations.

3. Symbol updates so that cross library calls can be directed to the correct location.

4. Rewrite of the updated shared library to a new file.

In the following sections implementation details and challenges of each process are discussed.

5.4.1 Profiling

In order to extract a list of functions that are usually not used by a program, we first observe several executions of the program and obtain a list of functions that are used by this program. A profiler is used to obtain a list of functions that are used at a specific run of the program. We combine all training runs and note all functions ever called.

There are various profilers that serve different needs. For our analysis we used *sprof* [26] and our own tool based on Dyninst [11]. *sprof* is a GNU profiling tool for shared libraries. We used this tool to create a quick mechanism to profile shared libraries, and used it for our initial tests. However, *sprof* can only profile one shared library per execution. Since this limitation impeded our simultaneous profiling efforts, we switched to our own Dyninst-based profiler. Our profiler rewrites shared libraries with instrumentation code and can profile all shared libraries at a single run.

5.4.2 Rewriting

Once the shared libraries are profiled, this profiling data is used to identify functions that will always be loaded, and those that will only be loaded on demand upon first call. Our system modifies LOAD entries in ELF headers so that the loader selectively loads functions that are known to be used. To maximize memory savings, functions are split into two groups: Used and Unused. Grouping functions requires moving around their machine code in the binary. Since the correct execution of an instruction usually relies on where it is located in binary (e.g. a relative jump instruction), extensive analysis is performed to make sure that the external behavior of an instruction does not change once it is moved. The shared library is then rewritten to the disk.

Binary code in static programs is not relocatable. Therefore, grouping unused functions in these executables would require modifying a large number of instructions to make them relocatable. Moreover, such executables are usually developed with one task in mind and unused functions occupy little memory. Therefore, these files are not modified: they are still loaded as a whole at runtime. Our tool links these driver programs with a shared library that is responsible for signal handling to load missing pages on-demand.

5.4.2.1 Avoiding Loading Unused Functions

To reduce the memory footprint of applications, we first need a mechanism that will allow us avoid loading parts of shared libraries while enabling the process to access these regions when necessary.

One way of getting around this problem is to split each shared library into two shared libraries: one that contains functions that will likely be called, and one that contains the remaining functions. If a function that is not currently available is called during runtime, a signal handling mechanism could load the shared library that contains this function and transfer control to that function. However, this scheme requires loading this whole shared library even though there is a single function that is used. As a result, it is not very effective in recovering from an unexpected call. Moreover, splitting shared libraries is complex since it requires moving most functions and symbols while regenerating the symbol table and procedure linkage table so that cross-library calls can be satisfied. It also requires adding a mechanism to access global variables across shared libraries.

Our mechanism, on the other hand, is very simple and effective. During the rewriting phase we modify appropriate ELF headers to accommodate selectively loading chunks of the original program or shared library. Loaders rely on program headers in ELF binaries and only load parts of the binary that has a matching LOAD entry in the program headers. Our rewriting mechanism modifies these LOAD entries by changing the address ranges and adding new ones if necessary so that it only loads desired (used) part of the library. The loader then loads the appropriate regions in the binary, leaving out the parts that are unlikely to be used.

5.4.2.2 Update Relative Calls

In a shared library with position independent code, most call instructions employ relative addressing. The exact target address is calculated using the current program counter and the offset that is stored in the call instruction. These offsets need to be updated during the function shuffling process if either callee or caller is moved. When the relative position of a call instruction changes with respect to its target, the address computation generates an incorrect address for the target unless the offset in the call instruction is correctly updated. As a result, our rewriting mechanism goes over every such instruction and updates the offsets to match the current layout.

5.4.2.3 Update Symbols

There is no guarantee that a given shared library will always be loaded at a specific address each time a process launches. As a result, addresses of functions in shared libraries cannot be known before launch time. Moreover, if a call instruction and its target are in different shared libraries, their relative position with respect to each other cannot be known before launch time. In such cases, any call instruction, rather than jumping directly to the callee, has to go through the procedure linkage table. The dynamic linker looks up for the callee upon the first call to that function. The look up process consists of matching the mangled name of the callee with a list of symbols that appear in the shared libraries. When a matching symbol is found, the address it contains is written to the corresponding procedure linkage table entry.

If a function is moved within a shared library without updating corresponding symbols, the dynamic linker cannot correctly look up for the actual address of this function since the symbol information still points to the old location of this function. Our rewriting mechanism locates such symbols and updates them with the new addresses of associated functions.

5.4.2.4 Update Jump Tables

Most current compilers make use of jump tables for n-way branches (e.g. switch statements in C). During the compilation process, such control flow structures are converted into an indirect jump instruction that reads addresses of targets from a table called jump table. In a shared library that contains position independent code, jump tables contain offsets rather than absolute addresses. These offsets correspond to the difference between the address of each target and the address of the global offset table of that specific binary.

If the function referenced in a jump table is moved, the jump table becomes invalid because the relative offsets of the function within the library have changed. Our rewriting mechanism updates each jump table entry for moved functions. We use Dyninst [11] to locate the jump tables for us since they are not marked by the compilers. An offset is computed such that it equals the displacement of the moved function from its old location in the shared library to its new location. That offset is added to each entry in the jump table associated with an indirect jump in this function.

5.4.2.5 Update Function Address Transfers

Function pointers are simply variables that contain addresses of functions. A function pointer becomes invalid when the associated function is moved to another address. Our tool recognizes writes to function pointers and updates them accordingly if the target functions are moved.

Since addresses of functions in shared libraries cannot be known before runtime, there has to be some runtime computation for writes to function pointers. There are two ways these addresses are computed:

1. Computed by the loader: Each function pointer that resides in the data section has an associated entry in the relocation table. The loader updates these function pointers during relocation. Moving a function invalidates the associated function pointer. Our tool checks each relocation entry and updates the ones that point to moved functions so that they will point to the correct location after relocation.

2. Using the global offset table address at run-time: In some cases, function addresses are computed using relative displacement of a function from the start of the global offset table. Moving the target function to another location requires updating this computation. This case requires thorough analysis since the address computation might take place at any valid code region. Therefore, an instruction-by-instruction analysis is performed to identify such computations. Once they are identified, offsets used in the address computation are updated.

5.4.3 On-demand Mapping

Our system provides a mechanism to recover when a function we did not load is called. As part of the offline analysis, the executable file is linked with a new shared library that contains a signal handler. During the execution if the control is transferred to some instruction that is not available in memory, the process generates a segmentation fault signal (SIGSEGV). Our signal handler, in turn, locates that function and maps it into memory (In reality, the whole page that contains this function is mapped since most systems only allow mapping an entire page). Systems with local hard drives can directly map a page to the memory. However, on HPC systems where compute nodes lack hard drives, load operations cannot be performed directly. On such systems, I/O nodes assist the compute nodes to load the required page over the network. Once the function is loaded, the execution resumes with the function that has just been loaded.

5.5 Experimental Results

To demonstrate our system, we performed our analysis on two sample PETSc applications (ex2 from the ksp package and ex5 from the snes package) [4] and GS2 [48, 25].

PETSc (Portable, Extensible Toolkit for Scientific Computation) is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It uses MPI for parallelization. It has linear and non-linear equation solvers and supports C, C++, Fortran and

Python. By default, PETSc generates statically linked libraries. For this work, we forced PETSc to create shared libraries as well. The first PETSc program we used, ex5 from the snes package, has 279 lines of code; whereas the second PETSc program we used, ex2 from the ksp package, has 79 lines of code. PETSc suite is composed of 879,772 lines of code in total.

GS2 is a physics application developed to study low-frequency turbulence in magnetized plasma. It is typically used to assess the micro-stability of plasmas produced in the laboratory and to calculate key properties of the turbulence which results from instabilities. It is also used to simulate turbulence in plasmas which occur in nature, such as in astrophysical and magnetospheric systems. GS2 is composed of 53,105 lines of code and is linked with a total of 20 shared libraries.

All shared libraries we examined were compiled with the debug flag on and without optimization. We used Open MPI [32] for an implementation of message passing interface.

5.5.1 Environment

We tested our system on a 64 node cluster owned and operated by UMIACS at the University of Maryland. Nodes are connected using Myrinet. Each node has two 32-bit x86 processors and runs Red Hat (version 4.1.2).

5.5.2 Results

Tables 5.1, 5.2 and 5.3 show how much saving one can achieve on a typical application. In our experiments the space savings of the text space ranged from

Table 5.1: PETSc results for *ex5* from snes package

Library Name	Original		Modified		Reduction	Reduction
	in Pages	in KB	in Pages	in KB	in Pages (%)	in bytes (%)
petsc	260	1034	68	266	73.85	74.24
petscdm	161	640	19	72	88.2	88.79
petscksp	335	1337	39	153	88.36	88.59
petscmat	772	3085	40	157	94.82	94.92
petscvec	204	813	52	205	74.51	74.76
petscsnes	20	77	20	77	0	0
mpi_cxx	10	36	5	16	50	54.93
mpi	142	564	37	144	73.94	74.45
open-pal	62	241	34	129	45.16	46.48
open-rte	55	215	34	131	38.18	39
m	28	108	3	8	89.29	92.27
X11	146	578	7	22	95.21	96.13
lapack	866	3458	2	2	99.77	99.94
blas	80	315	3	7	96.25	97.9
stdc++	133	529	12	45	90.98	91.54
gcc_s	12	45	2	5	83.33	88.95
Xau	2	3	2	3	0	0
Xdcm	3	7	3	7	0	0
gfortran	123	485	4	9	96.75	98.13
dl	2	4	2	4	0	0
nsl	14	55	2	7	85.71	87.59
util	2	2	2	2	0	0
TOTAL	2021	13632	348	1472	82.78	89.2

Table 5.2: PETSc results for *ex2* from ksp package

Library Name	Original		Modified		Reduction	Reduction
	in Pages	in KB	in Pages	in KB	in Pages (%)	in bytes (%)
petsc	260	1034	72	282	72.31	72.73
petscdm	161	640	3	8	98.14	98.75
petscksp	335	1337	49	193	85.37	85.56
petscmat	772	3085	49	193	93.65	93.74
petscvec	204	813	54	213	73.53	73.8
mpi_cxx	10	36	5	16	50	55.56
mpi	142	564	47	184	66.9	67.38
open-pal	62	241	37	141	40.32	41.49
open-rte	55	215	36	139	34.55	35.35
TOTAL	2001	7965	352	1369	82.41	82.81

34.6% to 100% for all shared libraries over 7 KB of text space. The total weighted average of space savings is 82.0%. There are some libraries that are used fairly often such as libopen-rte.so, which is a library in the Open MPI suite. On the other hand, some libraries such as libMdsLib.so are not used at all although they are linked with the application.

Figure 5.2 shows a comparison of running times between the original and modified applications in logarithmic scale. Although, the difference in running times is not large enough to make any conclusive statement, results support our assertion that our tool does not cause any performance overhead for applications that run more than a few seconds.

In our experiments, the modified GS2 runs took 5 seconds less than the un-

Table 5.3: GS2 results

Library Name	Original		Modified		Reduction	Reduction
	in Pages	in KB	in Pages	in KB	in Pages (%)	in bytes (%)
MdsLib	21	80	0	0	100	100
MdsShr	21	80	0	0	100	100
TdiShr	220	875	3	9	98.64	98.97
TreeShr	38	150	0	0	100	100
fftw	70	276	25	96	64.29	65.22
rfftw	58	228	8	28	86.21	87.72
mpi_f77	13	48	2	4	84.62	91.67
mpi	142	564	40	156	71.83	72.34
open-pal	62	241	36	137	41.94	43.15
open-rte	55	215	36	139	34.55	35.35
TOTAL	700	2757	150	569	78.57	79.36

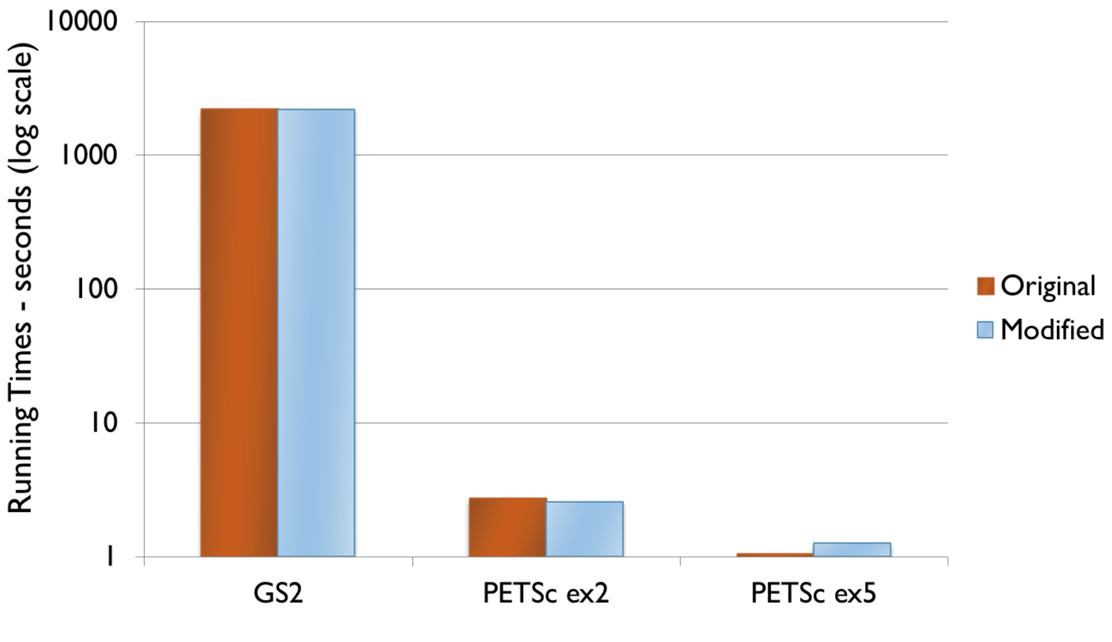


Figure 5.2: Running Times with Selective Program Loading in logarithmic scale

modified program (36 minutes 33 seconds vs. 36 minutes 38 seconds). Functions used by modified binaries occupy fewer pages; therefore, cache misses might be less frequent. Also, the paging system of the operating system might be spending less time loading and unloading pages.

On the other hand, applications that run for only few seconds might experience some slowdown due to initial signal handler registration. *ex5* from PETSc's *snes* package takes 1.05 seconds on average. In our experiments modified executable experienced about 19% slowdown since it did not run long enough to compensate for the cost of initial signal handler registration. Conversely, *ex2* from PETSc's *ksp* package runs for about 2.7 seconds and the modified executable experiences more than a 6% speedup. This result shows that this application runs long enough to compensate for the initial cost of running modified binaries.

Since most HPC applications take several minutes to complete, our impression is that modified binaries will likely not cause any overhead and might cause some speedup.

Just like any other on-demand tool, the performance of our system might suffer due to mapping. However, this operation is rarely necessary. In our experiments, the training runs accurately identified all functions that were called and thus on-demand mapping was never performed. Yet, we wanted to present the amount of time it takes to map one page into the memory at runtime. For this experiment, we created an executable with 100 functions where none of them were loaded into the memory at the start of execution. Since functions were aligned at page boundaries, these 100 functions occupy 100 pages. The executable file was stored on a remote server, and it was accessed through the network using the *SSH Filesystem* [77]. For this experiment, *SSH Filesystem*'s caching support was turned off. We ran this executable 5 times, forcing on-demand loading of missing pages a total of 500 times. Our experiments showed that loading a page into the memory over the local area network takes an average of 1.42 milliseconds. When the file is stored on a local drive, loading a missing page takes an average of 0.54 milliseconds. Loading a page once enables execution of all functions in that page. Moreover, with proper profiling, this operation is rarely necessary. Therefore, the cost of an occasional page load operation is negligible when the whole execution is taken into account.

5.6 Conclusion

In this chapter we proposed a new system that reduced memory footprint of executables linked with many shared libraries. After an offline rewriting phase, we managed to reduce the number of loadable pages in target shared libraries by an average of 85.0%. We also demonstrated that our tool causes no performance overhead for reasonably long running programs. Upon a call to a function that is not loaded into memory by the loader, our error recovery mechanism maps the page which contains that function into memory and continue execution without a failure, all in about 1.4 milliseconds. These properties show that our system performs a desirable optimization for frequently executed applications with multiple shared libraries.

Chapter 6

Security through Runtime Function Relocation

So far, we discussed ways to improve binary analysis and its uses. In this chapter, we show a different aspect of binary manipulation: improved security of applications. This work introduces a binary rewriting mechanism to make functions fully relocatable. Then, we show how these functions can be continuously relocated at runtime to improve the overall security of vulnerable applications. We show that programs rewritten by our mechanism are indeed more resistant against attacks. We also provide a discussion about the runtime overhead of our system, and conclude with a discussion of the security implications of our system.

6.1 Overview

Software security has long been a critical issue. Attackers exploit vulnerable software to steal unauthorized financial information, to obtain trade secrets, and, in some cases, just to have fun. Recent attacks carried out by the Stuxnet [50] and Flame [73] worms suggest that such attacks might also be carried out for cyberwarfare. In such an environment, cyber defense is a priority both for financial industry and national security.

In the center of successful system intrusion attempts, memory corruption vulnerabilities play a big role. Such vulnerabilities include buffer and integer overflows, format string errors, double-free problems, etc. Attacks that target these vulnerabilities rely on the knowledge of the locations of critical program elements.

It is now common knowledge that diversity in computer systems enhance overall security [31]. From an administrative point of view, though, diversity can be chaotic. To avoid problems with the administration of computer systems,there has been a tendency to provide security through launch-time randomization. The PaX Team was the first to introduce Address Space Layout Randomization [67]. This randomization process involves rebasing program code and data with an offset. Once the address space is randomized, an attacker has to try addresses from a large pool to find the location of a critical function. However, evaluating the effectiveness of Address Space Layout Randomization, Shacham et al. showed that brute force attacks are a big concern against services that fork many child processes [72]. In their experiments it took them only 216 seconds on average to obtain a remote shell on a target 32-bit machine. Evidently, just rebasing executable files may not provide the required level of security.

In this chapter, we present a novel approach for providing security through continuous randomization by relocating functions at runtime. Executable files are first rewritten to make individual functions completely relocatable. When the program is run, functions in that program and its shared libraries are relocated at runtime to prevent attackers from guessing the locations of these functions. This approach makes programs more resistant against attacks since attackers cannot guess the locations of critical functions. Even child processes that were once an exact copy of the parent process evolve over time and show different characteristics, reducing the chances of brute force attacks to succeed.

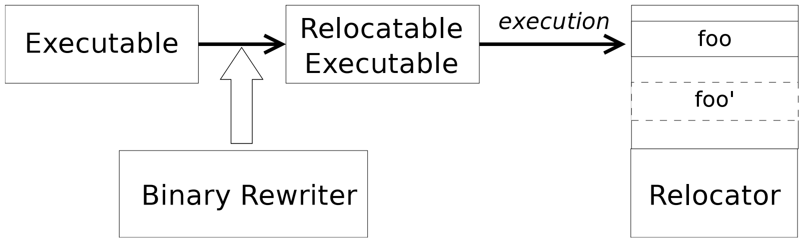


Figure 6.1: Overview of Runtime Function Relocation System

6.2 NINJA: Runtime Function Relocator

Our system, NINJA[1], provides runtime function relocation to prevent malicious attacks. Unlike previous systems where entire executable files are relocated once with a fixed offset for the entire duration of execution, our system continuously relocates individual functions at runtime using a user-defined relocation strategy.

Relocating a function is not as straightforward as just copying the bytes of that function; during relocation most of the addresses inside this function have to be updated, such as addresses in branch and call instructions as well as the ones in branch tables. This process requires analyzing the function and parsing its bytes to identify instructions and operands of these instructions. If this function is relocated again, all of these steps have to be repeated unless they are cached. Performing these operations at runtime causes a huge runtime overhead and is not practical.

To avoid the high cost of analyzing and updating functions at runtime, we analyze functions before execution and rewrite them to be completely relocatable, as illustrated in Fig. 6.1: Our binary rewriter takes in an executable file and produces a completely relocatable executable. Functions are relocated at runtime by our relocation module (aka Relocator) with no further modification to the relocated

[1]NINJA: **N**o **IN**trusion by **J**umping **A**round

function. This module first evaluates whether the function in question should be relocated with respect to a preselected relocation strategy; then copies the function to a new memory location, and updates associated data structures. Usually, only one memory location is updated during this process; the actual copy operation does not modify the bytes of the function.

In the following sections we explain how we make functions completely relocatable, and then describe how they are relocated by our relocation module.

6.3 Relocatable Functions

Normally, functions are not relocatable. They contain hard coded addresses that prohibit them from correctly executing at another location. Even functions that are part of relocatable libraries[2] are not individually relocatable because these functions use relative addressing across function boundaries, and they break when either the source or the target function moves to another location and thus change the relative distance between them. Therefore, *relocatable libraries* are only relocatable as a whole.

Our runtime instrumentation tool relocates a function after analyzing it thoroughly and rewriting the addresses it uses. This requires creating the control flow graph of that function, going over its instructions one-by-one and updating the ones that depend on addresses of other constructs. Moreover, any call instruction whose target is this relocated function has to be updated. It is worth mentioning that these call instructions can be anywhere in the program; therefore, all functions have to be

[2]Relocatable libraries are built with position independent code such as *gcc*'s -fPIC flag.

Table 6.1: Parts of Functions Modified by NINJA during Pre-processing

Function Entry	*Initial function location* is saved on the stack
Function Call	Function calls are redirected through the *Function Table*.
rsp-based Addressing	Offsets based on the stack pointer that point to parameters to this function are updated
rip-based Addressing	Addresses based on the instruction-pointer are modified to use the *initial function location* value instead.
Table-based Branching	Absolute addresses in branch tables are replaced with relative addresses. Current address is added to these offsets before branching takes place.
Function Exit	*Initial function location* is removed from the stack. Relocator is invoked if this function needs to be relocated.

analyzed to move even a single function. These operations are costly, and certainly unaffordable at runtime. To avoid this cost, we analyze functions before execution and rewrite them to be completely relocatable. For this purpose, our system heavily uses Dyninst's binary editing capabilities.

At the center of our approach is the Function Table, a table of function addresses. Each executable file has its own Function Table. Call operations read the addresses of target functions from this table. Another important mechanism towards relocatable functions is the store operation of a constant value on the stack. This value corresponds to the initial location of the function and is called *initial function location.*

Table 6.1 summarizes some of the actions taken for editing functions. Our binary rewriter updates all absolute addresses as well as all addresses based on the instruction-pointer. Moreover, call operations are rewritten to use the Function

Table. Since we store information about execution on the stack, some instructions that use the stack are also updated. In the following subsections we explain the steps taken while creating completely relocatable functions.

6.3.1 Function Calls and Function Table

A traditional call instruction contains the address of the target function. Relocating the target function, however, renders this call instruction invalid, as the control flow cannot reach the start of a function if it is not where it is supposed to be. Therefore, call instructions whose target is relocated have to be updated with the correct address. However, as we pointed out earlier, this would be extremely costly since such call instructions can be anywhere. To prevent this cost, each call instruction is rewritten before the actual execution to use the Function Table.

Each entry in the Function Table contains the address of the initial location of the function, its current location, its size, a counter of how many times it was called since last relocation, and a counter of how many frames on the stack correspond to this function. This last counter is required to avoid relocating a function when there is another active frame on the stack associated with it, such as in the case with recursive function calls. Otherwise, this relocation would cause a crash. When a call is made, the current location of the callee is read from this table and the call is carried out using this address. Each Function Table is stored in the data section of the associated executable.

The location of the Function Table remains fixed throughout the execution. Return-oriented programming attacks can only redirect control flow to a specific address, and forcing a jump to the Function Table will only result in a crash since the Function Table does not contain executable code. For a successful attack, an attacker first has to read the memory occupied by the Function Table, and then use the addresses stored in that table to successfully attack a target function. If an attacker can already read random memory in the address space of the attacked process, the program must have already been compromised, and further damage can be carried out by other means. The Function Table can be a source of vulnerability if an attacker manages to exploit a buffer overflow and overwrite the entries in the Function Table. These attacks can be prevented by storing the Function Table in pages of its own and using unmapped guard pages on each end of the table. Any attempt to overwrite entries in this Function Table will result in a crash when the attacker touches the unmapped guard pages. As a result, the fact that the location of the Function Table is fixed does not decrease the overall security of the application as long as it is not already compromised by other means.

6.3.2 Initial Function Location

When a source file is compiled into position independent code (PIC), compilers emit code with an addressing method based on the instruction pointer[3] and an offset representing the distance between the current instruction and the target address, instead of using absolute addresses. Usually, there are many occurrences of this addressing method in a function. When a function is relocated, offsets in each such instruction have to be updated since the value of the instruction pointer changes

[3]Instruction pointer is stored in the *rip* register on x86-64

due to relocation. However, we cannot afford to update every such offset during runtime function relocation.

To speed up this process, we rewrite each function in a way that offsets do not have to be updated during relocation. Instructions that use the instruction pointer are rewritten to use a constant value called *initial function location*, stored on the stack, and an offset computed during the rewrite operation. Since the *initial function location* value is constant, the offsets do not have to be updated. To accommodate shared libraries, this *initial function location* value depends on the actual location of the function when the program starts, calculated using the instruction pointer. When this function is relocated for the first time, the dependency on the instruction pointer is removed, and the *initial function location* is set to the address of the initial location of that function before the relocation. Hence, our addressing mechanism does not use instruction pointers, and offsets remain the same throughout the execution.

6.3.3 Indirect Branches That Use Tables

Compilers often convert switch/case statements into branches that use tables called jump or branch tables. These branch instructions read the target address from a table using an index calculated from the operand of the switch/case statement. These jump tables contain absolute addresses. When a function is relocated, the addresses stored in these jump tables become invalid. During the rewrite process, our system identifies jump tables and updates the entries in these tables to use offsets rather than absolute addresses. The offsets read from these tables are updated just before the branch operations take place using the current location of the function.

A similar operation is performed for indirect branches that deal with MOVAPS tables[4]. These branches are also rewritten so that the target address is updated using the current location of the function.

6.3.4 Accessing Parameters Stored on Stack

When there are more parameters than there are available registers for parameter passing, parameters are pushed to the stack before a function call takes place. Functions that are made relocatable by our rewriter also use the stack to store the *initial function location* value between the local variables and parameters passed to this function. Although local variables can still be accessed as before, accessing parameters stored on the stack requires us to modify the offsets used to address these stack locations. Parameters are usually accessed using either the stack pointer or the base pointer. Our rewriter analyzes all instructions that use these registers and updates their offsets if they access these parameters.

6.4 Runtime Function Relocation

Our system performs first-party runtime function relocation, i.e. executables are capable of and responsible for relocating their own functions, and there is no external monitor program. At the end of each function execution, we check how

[4]When a set of registers need to be saved before certain kinds of operations, a sequence of MOVAPS instructions are used. Depending on the parameters, the execution might save between 0 and 8 registers. Exactly how many of these registers will be saved are determined by an indirect branch.

many times that function has executed since the last relocation. Depending on our relocation strategy, our relocator decides whether we should relocate that function at that given time.

Assume *foo* is called. The call to the relocator takes place at the end of *foo*'s execution just before it returns back to the caller. However, any code after the call to our relocator will be inaccessible after *foo* is relocated since the old *foo* will be overwritten with zeroes. Our relocator, therefore, first relocates *foo*, then executes *foo*'s epilogue without giving control back to *foo*, and returns back to *foo*'s caller, by-passing *foo*'s own return instruction which has since moved.

The following steps are taken during relocation:

1. A chunk of memory that is large enough to store this function is obtained from the memory manager.

2. The bytes that make up the rest of the function are copied without any modifications.

3. The entry for the function in the Function Table is updated to reflect that function's new location.

If this is the first time this function is being relocated, the instruction that stores the *initial function location* value will be replaced by an instruction that pushes a constant value to the stack to be used as the *initial function location* as described in Sect. 6.3.2.

In the end, calls to this function will be redirected to the new location of the function with the help of the updated Function Table entry.

6.4.1 Relocation Strategies

Runtime function relocation is a costly operation. From a security point of view, relocating functions as often as possible is preferable; however, this frequent relocation incurs too much overhead to be practical. Therefore, we should be careful about when we perform function relocation. In our work we tried to find the balance between usability of our approach and the level of security we provide. In this section, we describe a continuum of relocation strategies our system supports:

1. **Always Relocate**: In this strategy, whenever a function returns, it is relocated. Our Relocator always relocates functions at the end of its execution. This strategy introduces a heavy overhead to program execution.

2. **Relocate After $\mathbf{n^{th}}$ call** Relocating functions after each execution is redundant for most functions, and not desirable in terms of overhead. In this strategy, functions are relocated after the n^{th} execution of that function. The number of relocations decrease by about n-fold, reducing the runtime overhead dramatically, even for n as low as 10, as we determined empirically (see Table 6.3).

3. **Relocate Randomly** In this strategy, relocation is performed periodically on a random function. When it is time to relocate a function, a separate thread picks a random function and relocates it. This strategy is fundamentally different from and orthogonal to the previous ones; and relocation can be performed on functions that are not yet called.

4. **Adaptive Relocation Strategy** Relocating functions too often may introduce a huge overhead. In this strategy, the frequency of relocation is determined on a per-function basis: Functions that are called too often are relocated less often per call than the functions that are called less often. To determine how often each function needs to be relocated, we first profile the application and gather a list of functions and how many times each of them was called. Functions that are called less than a threshold (we used 1,000 in our examples) are relocated every time they finish executing. Other functions are relocated after the n^{th} call, where n is picked with respect to how many times that function is called during our profiling runs and is in the [1 - 10,000] range. In our test environment all functions that execute more than 100 times and less than 10 million times are relocated a total of 100 to 1,000 times during the entire execution. Functions that execute less than 100 times are relocated every time they execute whereas functions that execute more than 10 million times are relocated every 10,000th time they execute.

6.4.2 Memory Management

When a function is relocated, it is copied into a new chunk of memory, and the old copy is dismissed. For a successful randomization effort, functions should be copied into random memory chunks. A naïve approach would be to allocate memory with *malloc*, and deallocate memory that holds the old location of the function with *free*. However, this approach tends to provide very poor randomization as malloc often returns recently *free*d memory.

We use a bucket approach to avoid the cost of allocating and deallocating memory too frequently, and to introduce consistent randomness. Each bucket is mapped to a random address in memory. Buckets can have one of 2^{24} addresses. Each bucket is 16 pages long and a function can be anywhere in a bucket, introducing another 16 bits of entropy. As a result, the total number of addresses a function may occupy is 2^{40}, i.e. NINJA provides 40 bits of entropy.

Our relocator initially allocates a fixed number of buckets. The first operation during relocation is picking a random bucket. If that bucket is full, it is removed from the available buckets list, and space for a new bucket is allocated to make sure there are at least a fixed number of available buckets. When an available bucket is found, a random gap is introduced. The function is copied right after this gap. The old location of the function is zeroed out. If the bucket that contains the old version of this function becomes empty, it is returned back to the list of available buckets.

We chose to provide 40 bits of entropy to match the amount of entropy promised by the PaX patch on Linux. It is possible to configure NINJA to provide more entropy by selecting the addresses of buckets from a wider range than 2^{24} unique addresses. Such a configuration decreases the likelihood of a successful attack.

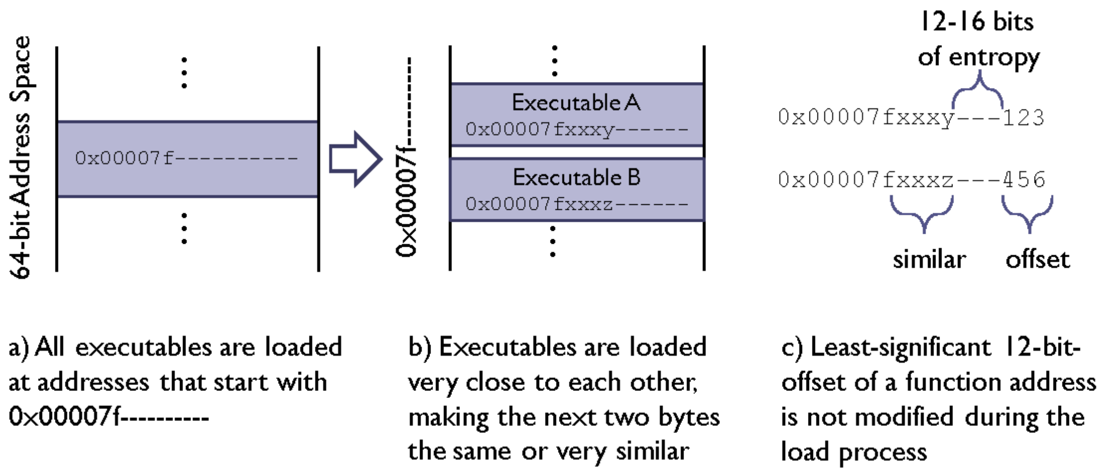
6.5 Security Implications of NINJA

Up to this point, we described how functions are relocated at runtime. In this section, we will explain how relocatable functions can prevent system intrusions originated by *return-to-libc* attacks.

In *return-to-libc* attacks, the attacker exploits a buffer overflow vulnerability by overwriting the return address stored on the stack. If the attacker knows the location of critical functions on the remote system, he/she can make the program *return* to one of these critical functions. This function in turn interprets values on the stack as parameters. For example, if the attacker makes the program *return* to the *system()* function[5] with correct parameters, he/she will be able to execute anything on the remote system accessible to the target process. It is not always simple to guess the location of a critical function; however, if an attacker has access to an exact copy of the program that is under attack (as in the case of open source applications and widely available commercial products), he/she can examine the program layout and determine the possible locations that a critical function might occupy. This process drastically simplifies the attack.

To make guessing the locations of critical functions harder, researchers introduced randomization techniques. These techniques shuffle locations of program components so that the critical functions might occupy different addresses across executions. What is shuffled and at what frequency depends on the technique used. The most common randomization technique is Address Space Layout Randomization (ASLR) [67]. Traditionally, ASLR loads shared libraries at random offsets at launch-time; therefore, functions inside these shared libraries are shifted to fixed, random addresses for the duration of the execution. The same technique can be applied to static executable programs through PIE (Position Independent Executable)

[5] *system()* function gathers the commands provided by the user through parameters and executes them on a shell.



64-bit Address Space
0x00007f----------
0x00007f----------
Executable A
0x00007fxxxy------
Executable B
0x00007fxxxz------
12-16 bits of entropy
0x00007fxxxy---123
0x00007fxxxz---456
similar
offset
a) All executables are loaded at addresses that start with 0x00007f----------
b) Executables are loaded very close to each other, making the next two bytes the same or very similar
c) Least-significant 12-bit-offset of a function address is not modified during the load process

Figure 6.2: Non-patched Linux Kernels Provide Only 12-16 bits of Entropy

mechanism. To date, executables are not created to be PIE by default: programs need to be compiled with specific compile flags to create PIE executables.

ASLR provides a maximum of 40 bits of entropy with patched Linux kernels. However, most Linux kernels lack this feature and thus ASLR can only provide 28 bits of entropy[6]. Since an application is only randomized once at launch time, a brute force attack is expected to take up to 2^{28} tries. Although ASLR rebases executables separately, a few of the most significant bytes of addresses remain the same for each executable as shown in Fig. 6.2: Without the enhanced ASLR kernel patch, the 24 high-order bits of all executables are 0x00007f. In our experiments, we realized the next 12 to 16 bits also tend to be the same. Therefore, 24 to 28 bits remain for randomization. Of these remaining bits, 12 of the low-order bits are not randomized as ASLR can only randomize executables at page boundaries. That leaves us with 12 to 16 bits of randomization. Figure 6.3 illustrates how an attacker can access a critical function by just guessing a few bytes and reusing the

[6]Some distributions such as Alpine Linux and Hardened Gentoo supports a patched version of the Linux kernel out of the box.

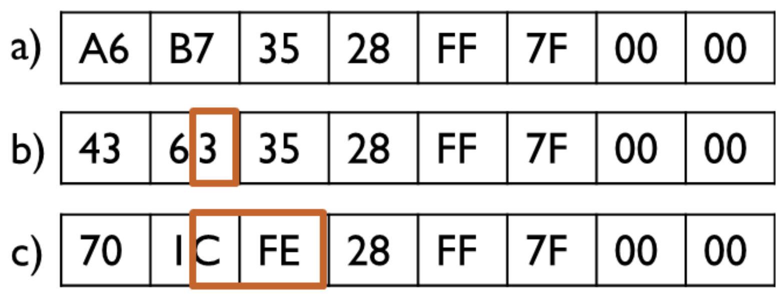


Figure 6.3: Reduced entropy on little-endian architectures: a) Original return address. b) Target is close to original return address. c) Target is far from the original return address yet a number of the most-significant bits are the same.

most significant bytes on a little-endian architecture, e.g. the x86. Figure 6.3 a) shows the original return address. If the target function is close to the original return address, as is the case in Figure 6.3 b), then only guessing bits 13 through 16 correctly is enough to get to the target function (The least significant 12 bits can be obtained by analyzing the disk image of a copy of the executable). If target function is further away, the number of different bits that have to be overwritten increases. However, in most cases, the most significant bits remain the same, as demonstrated in Figure 6.3 c). Being able to guess the address of a target function by just overwriting few of the least significant bits drastically reduces the search space.

Apart from the above weakness due to limited entropy, ASLR is also extremely vulnerable when a process keeps launching child processes during execution as the Apache HTTP Server does. Since each forked child retains the same memory layout as the parent process, the attacker practically has an infinite number of processes with the same memory layout to attack. Eventually, the address of a critical function will be exposed.

Our system, on the other hand, relocates functions both post-link-time and at runtime. Our system first shuffles functions, then rewrites them to be fully relocatable. Every rewritten executable has a different layout. Even if an attacker has a copy of the original executable, he/she cannot easily guess where each function is located after the rewrite. After the program is launched, functions are relocated randomly, giving the executables a different layout every time a function is relocated. This relocation may take place as often as after every function execution. Even if the attacker has access to an exact copy of the target executable after the rewrite, he/she will not be able to identify function start addresses at runtime, as they change continuously. If the attacker somehow manages to find the location of a function, that function will be relocated soon, and the attacker may not be able to attack that function in time. Since functions are randomly relocated, a parent process and all of its child processes will all evolve into having separate memory layouts during execution, making it much safer to fork new processes during execution.

6.6 Security Evaluation

To show the effectiveness of our system from the security point of view, we experimented with a modified version of the Apache HTTP Server[30]. It is developed and maintained by The Apache Software Foundation and is written in C. It is composed of more than 400,000 lines of code. The modification we applied to this process is the introduction of a simple bug that can be exploited via an HTTP request. Using this bug, we tried to perform a return-oriented attack to execute a function that was not supposed to execute at that point during execution. We launched our attack from a client that issues HTTP requests. In our experiments, we assumed the attacker had an exact copy of the executables that we experimented with.

The result of our experiments is given in Table 6.2. In our first set of experiments, we attacked *httpd* executables that are only protected by the ASLR. For the non-PIE *httpd* executable without runtime function relocation, we consistently succeeded in executing a particular target function, *foo*, that resides in the *httpd* executable (i.e. not in a shared library) in our first attempt during all of our 5 experiments by exploiting the bug we introduced. Since the address of the target function can be obtained by looking at the symbols in the *httpd* executable, our attack did not involve any guessing. Then, we attacked the PIE version of the *httpd* executable. Attacking the same target function, *foo*, succeeded almost instantaneously using the technique described with Fig. 6.3, as we only had to perform 16 attacks. Attacking another function, *bar*, that resides outside the *httpd* executable, namely in *libapr-1.so*, took more time as we had to try more addresses. Since ASLR cannot really provide too much entropy even for shared libraries, we were able to have *bar* executed in 4.9 hours on average. We believe this time frame is short enough for successful attacks on systems that are not constantly monitored.

Then we tried attacking the *httpd* and *libapr-1.so* executables fortified by NINJA. In this case, we also tried to benefit from the little-endianness of the architecture. However, since functions are randomly spread in the memory, we observed exploiting little-endianness of the architecture did not provide any solid benefits.

Table 6.2: Time Required for a Successful Attack with/without Runtime Function Relocation

	Non-PIE	PIE - same exec	PIE - library
ASLR	Instant (single try)	Instant (multiple tries)	4.9 Hours
NINJA	$>$ 24 Hours	$>$ 24 Hours	$>$ 24 Hours

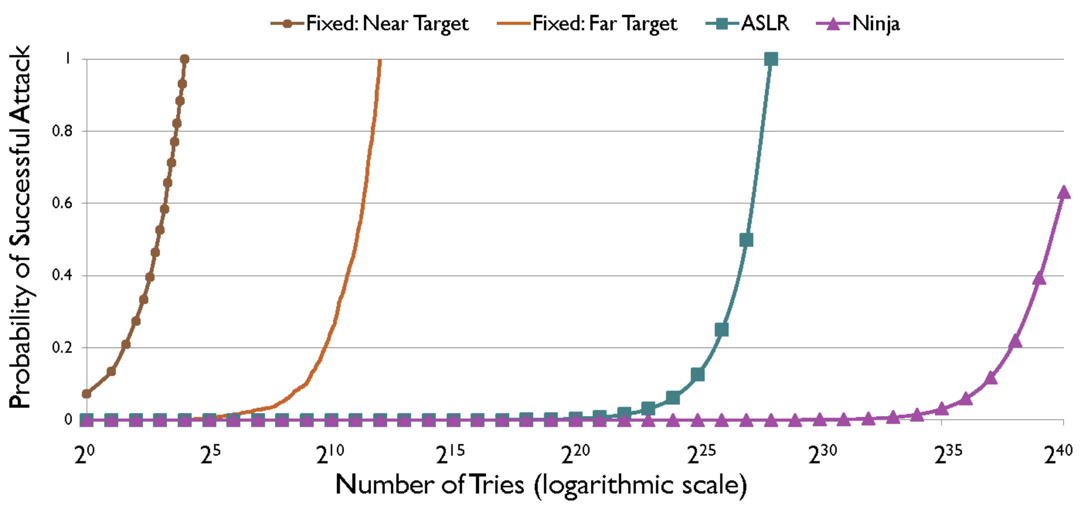


Figure 6.4: Probability of a Successful Attack with ASLR and NINJA (logarithmic scale)

With this approach, whether the target function is in the same executable or not, or whether the executable file is PIE or not did not matter as functions were relocated by NINJA continuously in all three cases. For each run of our attack, we stopped our experiments when the elapsed time reached 24 hours.

We show expected and measured probabilities of successful attacks with ASLR and NINJA in Fig. 6.4. The rightmost curve shows the probability of a successful attack on a system fortified with NINJA. As continuous function relocation prevents any gain brute-force attacks may provide, this curve basically matches the probability of a successful attack in a search space of 2^{40} different addresses with replacement. The model we used to calculate the probability of a successful attack on NINJA is given in Sect. 6.7. The second curve from right, *ASLR*, shows the probability of a successful attack with ASLR that provides 28 bits of entropy, the default setting for ASLR on unpatched kernels. However, in practice, ASLR can be beaten with far less tries on little-endian architectures as described in Sect. 6.5. The third curve from right, *Far Target*, shows the probability of a successful attack on little-endian architectures as obtained by our attack simulation[7] on system ASLR when the target function is far from the original return address. Finally, the leftmost curve shows the probability of a successful attack when the target function and the original return address are close to each other, e.g. they are in the same executable file. This type of attack takes a maximum of 16 tries, e.g. an attacker only needs to guess 4 bits.

6.7 Model of NINJA Security

In this section we provide a model that shows the enhancement of security when two of the relocation strategies described in Sect. 6.4.1, *Relocate After n^{th} call* and *Relocate Randomly*, are applied together.

Assume it takes a function on average E seconds to execute as the leaf function on the stack,and assume the relocation threshold, n, is set to N. On average, it takes EN seconds for a relocation to occur with this strategy. Therefore: ${}^{1}/_{EN}$ relocations occur in one second. Moreover, a random function is relocated every R seconds. As

[7]Our simulation calculated the number of tries to successfully attack the Apache HTTP Server on a real unpatched Linux system. For each iteration of our simulation, we launched the Apache HTTP Server, obtained the addresses of target functions and original return addresses, and fed these addresses to our simulator to calculate the number of tries required to guess the target address on the system. Note that an actual brute-force attack would take exactly the same number of tries.

a result, ${}^{1}/_{R}$ relocations take place due to our random relocation strategy. The total number of relocations that take place in one second is then given by:

$$\frac{1}{EN} + \frac{1}{R} = \frac{R + EN}{REN} \tag{6.1}$$

If a single attack takes T seconds, and if there are F functions in the executable, the probability of the relocation of a specific function between two attacks is:

$$P_{reloc} = \frac{T(R + EN)}{FREN} \tag{6.2}$$

A successful attack takes place when i) the target function is at the location that will be attacked next, and if it does not get relocated, ii) the target function is relocated to the address that will be attacked next. The total probability of a success at a given try is:

$$P_{atck} = (1 - P_{reloc})\frac{1}{A} + P_{reloc}\frac{(A-1)}{A}\frac{1}{A} = \frac{1}{A} - \frac{P_{reloc}}{A^2} \tag{6.3}$$

Here, the first term shows the probability of the function being at the address that will be attacked next (${}^{1}/_{A}$) and the probability of that function not being relocated ($1 - P_{reloc}$). The second term, on the other hand, shows the probability of that function being relocated (P_{reloc}), the probability of the function being at another address (${}^{(A-1)}/_{A}$), and the probability of that function being relocated to that address (${}^{1}/_{A}$).

The likelihood of a successful attack during a series of attacks is:

$$P_{atck} + P_{atck}(1 - P_{atck}) + P_{atck}(1 - P_{atck})^2 + \ldots = P_{atck} \sum_{i=0}^{\infty} (1 - P_{atck})^i \tag{6.4}$$

This is the geometric series. The probability of a successful attack in the first n tries is then given by:

$$P_{atck} \sum_{i=0}^{n-1} (1 - P_{atck})^i = P_{atck} \frac{1 - (1 - P_{atck})^n}{1 - (1 - P_{atck})} = 1 - (1 - P_{atck})^n \tag{6.5}$$

The dominant term in P_{atck} turns out to be ${}^1/_A$, the probability of a successful random attack. This observation shows that the probability of a brute force attack is almost identical to the probability of a random attack, even slightly lower. Therefore, a brute force attack will not increase the chances of a successful attack.

In our experiments, we observed that sweeping the whole 28-bit search space ASLR provides takes about 200 days (about 16 addresses per second). Although an uninterrupted attack that lasts 200 days is somewhat unlikely, the fact that there is a .5% chance that ASLR can be defeated within a day shows that ASLR alone is not very reliable. Moreover, in our experiments, we showed that we can have a critical function executed in less than 24 hours in all of our attempts without exhausting the full address space. NINJA decreases the likelihood of a successful attack in a single day to 1.22e-4%, i.e. to about 1% of a 1% of a 1%. The comparison is illustrated in Fig. 6.5. With NINJA, the probability of a successful attack is much less than that of ASLR.

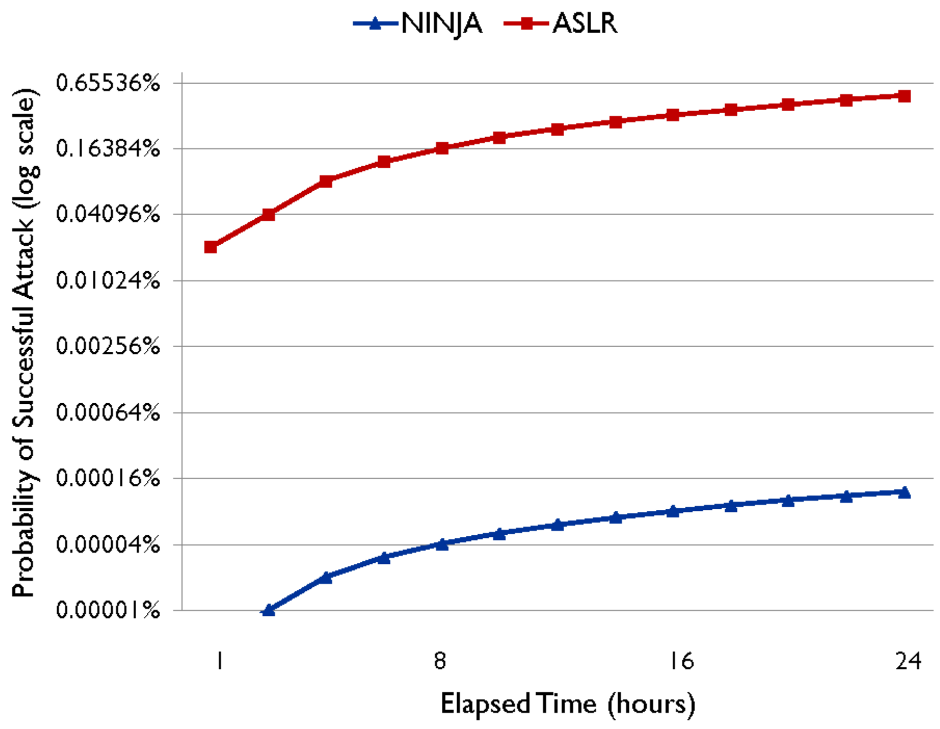


Figure 6.5: Likelihood of a Successful Attack in 24 Hours

Information leakage is a big concern for ASLR: When the location of a function or a known offset from a function is leaked, the security ASLR provides is compromised. NINJA, on the other hand, relocates functions continuously: Even when the location of a critical function is revealed, that function will soon be relocated by NINJA, and attacks that follow will fail. The time it takes for a specific function to relocate is 750 seconds when $N = 100$ and $R = 1$. When relocation is performed more frequently, as when $N = 1$ and $R = .1$, it takes a given function about 14 seconds to relocate. Figure 6.6 shows the average duration a function occupies the same address with various N and R values. Note that the smaller the duration, the sooner that function will be relocated by NINJA.

To demonstrate increased security offered by NINJA against brute force at-

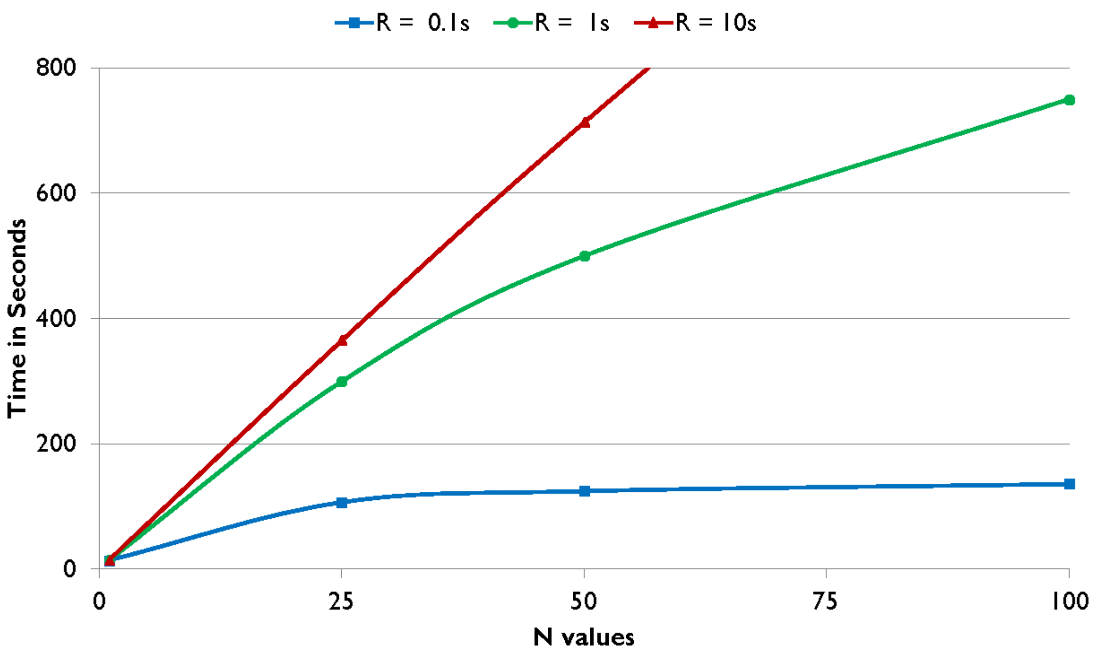


Figure 6.6: Duration a Function Occupies the Same Address, in Seconds

tacks, Fig. 6.7 shows the variation in the probability of a successful attack with respect to the number of tries on a given set of relocation parameters. Although NINJA normally provides 40 bits of entropy, we pretend we can only use 16 bits of entropy for this graph to be able to show the variation more clearly. This graph verifies our expectation that relocating functions more often provides a higher level of security.

6.8 Performance Evaluation

To evaluate the overhead of runtime function relocation, we applied our technique on benchmarks in SPEC CINT2006 [38, 61]. SPEC CINT2006 contains a series of CPU-intensive executables that are selected to evaluate the processor(s) and the memory system. All together, SPEC CINT2006 has about 1,047,000 lines

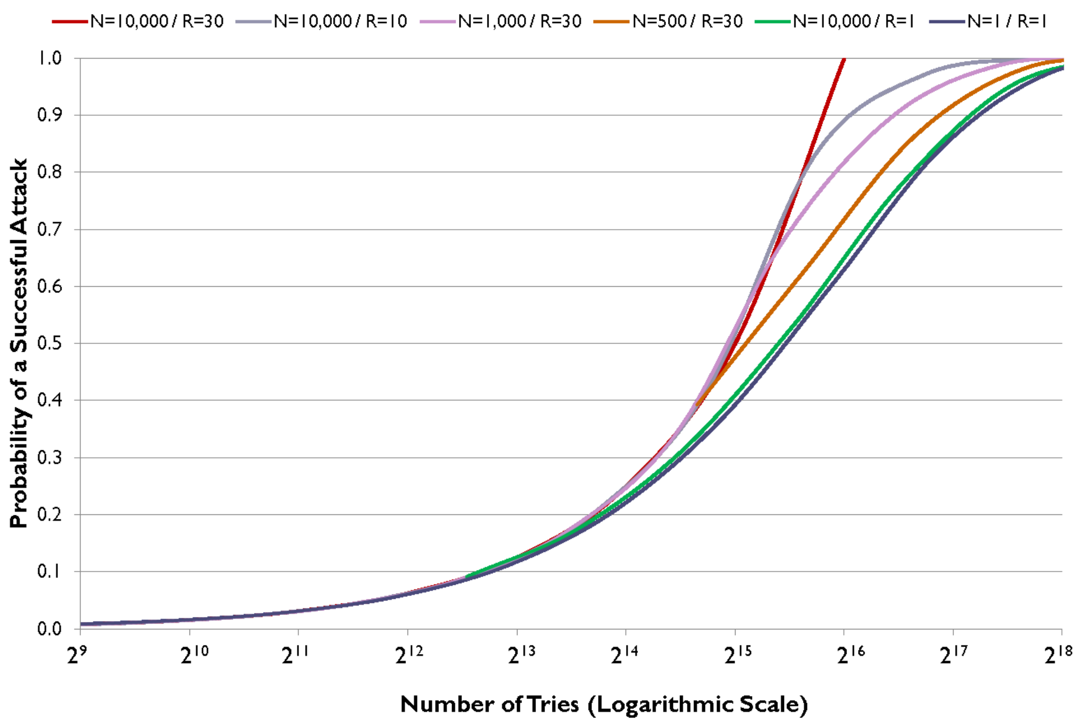


Figure 6.7: Probability of a Successful Attack with Varying Relocation Parameters

of code. We selected benchmarks written in C to evaluate the overhead of our system. We ran each set of benchmarks on a 64-bit Linux machine with 2 Intel Xeon processors with 6-cores each, clocked at 2.53 GHz, with a total main memory of 48GB.

Table 6.3 shows normalized running times of SPEC CINT 2006 benchmarks with runtime function relocation at varying frequencies. The leftmost column, *none*, shows the overhead of executing rewritten binaries without any runtime relocation, demonstrating the runtime overhead introduced by our edits described in Sect. 6.3. The columns that follow show the overhead of applying function relocation at runtime: 10,000 means that functions are relocated every 10,000th time they are executed.

Relocating functions often is clearly very costly. To improve the performance

Table 6.3: Runtime Overhead of Runtime Function Relocation on SPEC CINT 2006

Benchmark	None	10,000	1,000	100	10	1	Adaptive
perlbench	1.50	1.61	2.52	3.15	15.17	119.39	1.62
bzip2	1.01	1.03	1.16	1.18	2.49	13.44	1.02
mcf	1.07	1.18	1.44	1.05	3.64	25.45	1.21
gobmk	1.10	1.15	1.42	2.09	8.99	70.05	1.15
hmmer	1.00	1.00	1.06	1.01	1.08	1.69	1.00
sjeng	1.28	1.33	1.73	2.17	7.52	59.38	1.32
libquantum	0.98	1.01	1.06	1.03	1.24	3.37	1.21
h264ref	1.61	1.64	2.19	2.11	6.26	46.47	1.65
AVERAGE	1.19	1.22	1.51	1.61	4.79	33.75	1.25

of runtime function relocation, we profiled these benchmarks and created a custom relocation period for each function using a simple formula. In our approach, any function that executes less than 1,000 times is relocated every time it is called. Most of the remaining functions are relocated 100 to 1,000 times during the execution of the benchmark. Some functions which are called very frequently are relocated every 10,000th time, and they are relocated more than 1,000 times during the execution of the benchmark. We call this technique *Adaptive Relocation*. Table 6.3 also shows the overhead associated with this *Adaptive Relocation* strategy. This strategy proves to be the most useful since it offers frequent relocation with a low overhead.

Figure 6.8 illustrates the effects of applying execution-based and time-based relocation techniques on the performance of benchmarks from the SPEC suite. We observe the least overhead when $N = Infinite$ and $R = Infinite$, i.e. when no relocation is performed. Using smaller N values increases the frequency of relocations

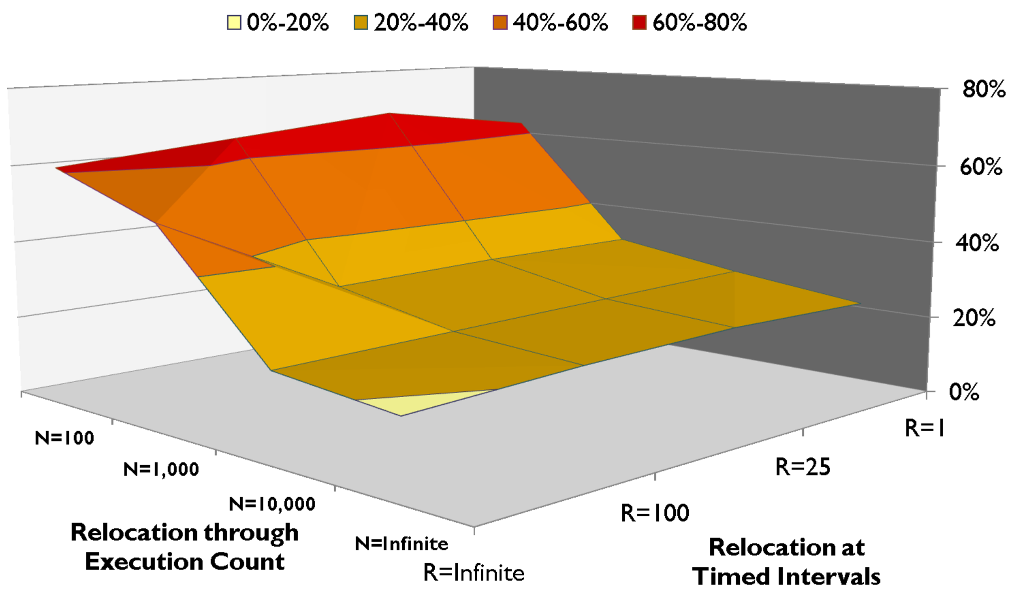


Figure 6.8: Relocation Overhead with Varying Relocation Frequencies

at the expense of increased runtime overhead. Using lower R values also tends to increase the runtime overhead due to the increased relocation frequency. However, performance effects in this case is not observed to be as dramatic.

6.9 Conclusion

In this chapter, we introduced a new security mechanism that reduces the likelihood of successful attacks through continuous function relocation at runtime. Our system first rewrites executable files to be completely relocatable, linking them with a relocator library. When we determine a function needs to be relocated, we just copy the function to randomly selected vacant memory and update the corresponding entry in the Function Table accordingly.

Our system does not require a change in the software distribution. Off-the-shelf executables can be secured by system administrators after they are installed. Runtime relocation is performed with no additional user interference.

We showed that our system dramatically reduces the likelihood of a successful attack. Since locations of functions continuously change, malicious users can only try random attacks. Unlike traditional randomization techniques, even child processes that were once an exact copy of the parent process evolve over time by relocating functions to different locations in the memory. Attackers do not get any advantage over the single process case. Our system greatly increases the difficulty of attack in that respect compared to traditional randomization techniques where child processes have identical layouts as the parent processes, and attackers get an arbitrarily large number of processes with the same layout to attack.

Our system inevitably introduces some runtime overhead. In our experiments we showed the overhead was observed at 25% on average, but varies with the user controllable frequency of relocations. Our technique is able to change the ASLR defeat success chances for a vulnerable HTTP server from 0.5% in 24 hours of attack to 1.22e-4% chance of success in the same 24 hour interval. NINJA can be configured to provide more than 40 bits of entropy to reduce this probability even further.

Chapter 7

Future Work

Over the course of our work, we identified some extensions to our current research. These extensions are described in our short-term road map. Moreover, we determined new directions our research can take, as described in our long-term road map.

7.1 Short-term Road Map

Our short-term road map describes direct extensions to our current research. Each of these projects are expected to take 4 to 6 months to complete.

7.1.1 Compile-time Support for Function Relocation

We demonstrated the benefits of using a mechanism to rewrite binaries with fully-relocatable functions. However, analyzing and rewriting binaries is not the best way to generate fully-relocatable code. We believe this work should be delegated to the compiler, and binary rewriting for this purpose should only be used when recompilation is not possible. Besides, a compiler may be able to allocate registers more efficiently whereas a rewriter can only use a register after saving the original value and restoring that value back at the end of the generated code, unless the register in question is dead.

7.1.2 Compile-time Support for Reducing Memory Overhead

Compiler support might also be beneficial in reducing the memory overhead of applications when there are many unused functions inside these applications. Given an execution profile, many compilers know how to optimize the generated executable when the code is recompiled. Using these profiles, compilers can better place functions to increase the instruction cache performance. Compilers can also be modified to place functions in groups so that functions that are unlikely to be used can be put on the same page(s). These pages can then be left out at the loading phase, reducing the memory footprint of the target applications. This approach eliminates the extra pass performed by our rewriting operation.

7.1.3 Secure Portable Devices

Portable devices have become quite common with the introduction of smart phones. Consumers store information about their email and bank accounts on these devices. Moreover, they are also increasingly being used for business activities, replacing Point-of-Sale equipment. The main drawback of these devices is that they have limited computational power. Continuous function relocation or similar strategies may not be practical for such devices as these operations would drain the battery rather quickly. Our approach for securing such machines is to apply function relocation only to critical functions. Since our current approach supports function level granularity, our system can be extended to portable devices.

7.1.4 Analyzing Effects of Function Relocation to ICache and TLB

As an extension to our work, we would like to investigate the effects of runtime function relocation to instruction cache and TLB performance. In our system, functions are relocated as a whole without changing the control flow. Therefore, we expect the effects of runtime function relocation to the instruction cache to be negligible. On the other hand, functions are spread across many pages. Whenever a function is accessed in a page that is not in the TLB, a TLB miss occurs. The larger the number of pages used by the program, the higher the probability of a TLB miss. An analysis of TLB performance might be important for later uses of runtime function relocation.

7.1.5 Post-link-time Function Inlining

Function inlining is performed by compilers during code generation to avoid costly function calls when the advantage of having a separate implementation of that function is very limited. An inlined function is executed without a function call, and optimizations on registers can be carried out from the encapsulating function to the inlined function. However, functions cannot be inlined when their implementation is not available during compilation. Such cases frequently happen when the implementation of a function is compiled into a different object file than the file into which caller of that function is compiled. All of the functions except for the ones that are in a shared library are linked together into one executable. At this point, the implementation of a called function can be inserted into the caller function by a post-link-time binary rewriter. We can use our expertise in binary analysis and rewriting to provide a tool that supports post-link-time function inlining.

7.2 Long-term Road Map

In this section, we present research ideas to which our experience with runtime code generation and program modification can be applicable. These areas include virtual machine migration, execution migration, and runtime code generation for auto-tuning. We expect each of these ideas to take 12 to 18 months.

7.2.1 Live Virtual Machine Migration

In this dissertation, we presented techniques to relocate functions at post-link-time and at runtime. A natural extension of our work is to apply runtime migration techniques to processes. However, moving a process from one execution environment to another might be problematic if the underlying systems differ.

One abstraction to remove complications of system specific execution environments is virtual machines. Virtual machines hide the details of underlying systems and provide an identical execution environment to processes even when they run on different architectures. They also provide sandboxing so that a process can only affect the state of the virtual machine that it runs in. Moving the virtual machine to another physical machine preserves the integrity of processes running on that virtual machine. As part of our future work, we plan to tackle the challenges of relocating a live virtual machine to a different execution environment while still supporting interactive applications that run on it.

7.2.2 Decoupling Execution from Physical Mediums

Tomorrow's computation devices will be tiny and gigantic. We already use pocket-size portable computers in the form of "smart phones" and connect to cloud services for most non-trivial tasks. We expect this trend to continue, dramatically increasing the differences in size and computation power between the systems that provide services and the end-user devices that display the results.

Currently, portable devices only carry out light-weight tasks, mostly to save bandwidth and to avoid paying the round-trip cost of accessing a remote server. Using an adaptive execution strategy benefits users in terms of both response time and availability. If wireless connectivity is not possible, more tasks should be performed on the portable device. If wireless connectivity is possible, tasks can be migrated to remote servers, and should be done so more liberally when the load on the portable device is high. For seamless execution, this migration has to be quick and efficient.

The virtual machine migration we discussed in the previous section may also benefit portable devices. Smart phones run applications in sandboxes that can be copied to another execution environment at runtime, provided that the new execution environment can emulate the previous execution environment. Any application that runs on such a sandbox can then continue execution at this new environment. This approach can be useful to move execution from a portable device to a server, or from one portable device to another one (e.g. to a device with a bigger screen).

Another way to achieve this migration is to develop common software development frameworks for both consumer electronics and large scale systems. Users can pick how much work should be done on their devices before migrating tasks to back-end servers by choosing threshold values, either manually or through an adaptive process. If execution should be migrated, current state of execution will be transferred to remote system and execution will resume there. Results will be returned and displayed on user's device once the operation is complete.

7.2.3 Use of Hardware Performance Counters for Runtime Code Modification

Processors are equipped with a set of registers called *hardware performance counters* that can be programmed to count specific events that take place during execution. Some of the supported operations include counting the number of instructions that have executed, the number of cache misses at various levels of the cache hierarchy, or the number of branches. The values stored in these registers can be read at runtime; therefore, application developers get a chance to tune their programs at runtime.

Tuning of applications has been studied extensively in recent years. Researchers have come up with mechanisms to support auto-tuning of programs at runtime. The state of the art in auto-tuning is to measure the performance of a given set of parameters and search the parameter space to find optimal configuration. Hardware performance counters, however, can provide feedback to search the parameter space in the correct direction. For example, if the application experiences a high number of L1 cache misses, our search algorithm could try to limit the loop-unrolling factor. With this approach, hardware performance counters may provide auto-tuners a better search strategy with faster convergence.

Chapter 8

Conclusion

In this dissertation, we introduced new techniques for improving flexibility, parsing speed, memory footprint, and security of executable files. We evaluated our techniques on real life applications and full-size benchmarks and showed that our techniques can be adapted without considerable drawbacks.

First, we presented *Relocatable Basic Blocks* that can be moved around individually without modifications to the rest of the system. Applications that benefit from Relocatable Basic Blocks include *Basic-block Linkage Table*s, and in some cases *Target Address Table*s. Moving these basic blocks only requires copying the code to new memory and updating corresponding entries in these tables. *Relocatable Basic Blocks* can be used for a simplified runtime instrumentation approach. However, overhead of using these basic blocks can sometimes hinder the benefits of using them for increased flexibility.

We then introduced a new compilation mechanism for improved binary parsing speed. Our compilation mechanism intercepts calls to the system compiler using a wrapper, and creates executable files with pre-computed Control Flow Graphs (CFGs) stored inside the binary. These CFGs are computed at the assembly instruction level and are accurate when the executable file is generated. They are stored in a section that is not loaded into the memory at runtime; therefore, the running time and memory footprint of applications compiled with this compilation infrastructure are not affected.

Any binary analysis tool can benefit from using these tables to build CFGs. To demonstrate their effectiveness, we modified Dyninst [11] binary analysis tool to read in the information about CFGs. This approach drastically sped up the process of building CFGs: we observed an average speed-up of 3.8x with a maximum speed-up of 4.4x.

Another mechanism we introduced targets reducing the memory footprint of code segments of shared libraries. This mechanism involves profiling shared libraries for common use cases, then identifying functions that are not used in these common cases, and grouping them together. Afterwards, our binary rewriting mechanism modifies the program headers in these shared libraries to make the loader skip these unused functions during the loading of these shared libraries into the memory. If a function that was deemed unused is called during execution, our on-demand function loader finds that function on the disk, and loads the page that contains that function into memory.

Our mechanism that avoids loading unused functions of shared libraries is able to reduce the number of pages occupied by the code segments of shared libraries by 85.0% on average. Sometimes the code section of entire libraries need not be loaded into memory (i.e. 100% reduction in number of pages that contain the library). In our work, we also showed that our tool does not introduce any significant performance overhead for programs that run for at least a few seconds.

Finally, we show that applications enjoy greater security when their functions are relocated at runtime. In this work, we first rewrite executable files to make their functions completely relocatable, then link them with a relocation library that is responsible for moving functions at runtime. Relocation takes place either on random functions at timed intervals or at the end of the execution of a function when that function has executed more than a specific number of times. We present a mathematical model that shows the likelihood of a successful attack is reduced when our technique is applied to executables. Moreover, our experimental results show that executables enhanced by our tool are more secure than regular executable files: Remotely forcing execution of a target function in a regular executable file takes 4.9 hours on average whereas we were not able to have that same function executed in a fortified executable in 4 days during any of our experiments. Although this approach introduces some runtime overhead - 25% on average with the default settings - factors that cause this overhead are adjustable, and the overhead can be reduced to about 19%.

In conclusion, this thesis shows that binary rewriting improves memory performance of applications and increases overall security of critical functions while also demonstrating modified compilation mechanisms can assist in making binary analysis easier.

Bibliography

[1] N. Aaraj, A. Raghunathan, and N. Jha. Dynamic binary instrumentation-based framework for malware defense. In D. Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 64–87. Springer Berlin / Heidelberg, 2008.

[2] N. R. Adiga et al. An overview of the bluegene/l supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[3] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis. Defending against hitlist worms using network address space randomization. In *Proceedings of the 2005 ACM workshop on Rapid malcode*, WORM '05, pages 30–40, New York, NY, USA, 2005. ACM.

[4] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[5] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.

[6] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev. Address space randomization for mobile devices. In *Proceedings of the fourth ACM conference on Wireless network security*, WiSec '11, pages 127–138, New York, NY, USA, 2011. ACM.

[8] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis. On the general applicability of instruction-set randomization. *IEEE Trans. Dependable Secur. Comput.*, 7:255–270, July 2010.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[10] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In R. Buschkes and P. Laskov, editors, *Detection of Intrusions and Malware & Vulnerability Assessment*, volume 4064 of

Lecture Notes in Computer Science, pages 129–143. Springer Berlin / Heidelberg, 2006.

[11] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14:317–329, November 2000.

[12] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *SIGPLAN Not.*, 29(11):242–251, 1994.

[13] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 188–, Washington, DC, USA, 1997. IEEE Computer Society.

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

[15] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, New York, NY, USA, 1994. ACM.

[16] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded risc processors. *SIGPLAN Not.*, 34(5):139–149, 1999.

[17] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pages 346–355, 1998.

[18] C. Curtsinger and E. Berger. STABILIZER: Enabling statistically rigorous performance evaluation. Technical report, University of Massachusetts, Amherst, 2012.

[19] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27(5):882–945, Sept. 2005.

[20] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA*, pages 1013–1019, 2000.

[21] S. Debray and W. Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, New York, NY, USA, 2002. ACM.

[22] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, 2000.

[23] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, 2000.

[24] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 257–268, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[25] W. Dorland, F. Jenko, M. Kotschenreuther, and B. N. Rogers. Electron temperature gradient turbulence. *Phys. Rev. Lett.*, 85(26):5579–5582, Dec 2000.

[26] U. Drepper. Using elf in glibc 2.1. Technical report, Cygnus Solutions, Sunnyvale, CA, 1999.

[27] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[28] M. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 27 – 36, nov. 2004.

[29] H. Etoh and K. Yoda. propolice: Improved stack-smashing attack detection. *Transactions of Information Processing Society of Japan*, 43(12):4034–4041, 2002.

[30] R. Fielding and G. Kaiser. The apache http server project. *Internet Computing, IEEE*, 1(4):88–90, 1997.

[31] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67 –72, Los Alamitos, CA, USA, may 1997. IEEE Computer Society.

[32] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[33] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 40–40, Berkeley, CA, USA, 2012. USENIX Association.

[34] GNU Project. GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF). http://gcc.gnu.org/. Retrieved: June, 2013.

[35] Google, Inc. V8 benchmark suite - version 7. http://code.google.com/p/v8/. Retrieved: March, 2012.

[36] H. Guo, J. Pang, Y. Zhang, F. Yue, and R. Zhao. HERO: A novel malware detection framework based on binary translation. In *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, pages 411 – 415, October 2010.

[37] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33:63–68, December 2005.

[38] J. L. Henning. Guest editor's introduction. *SIGARCH Comput. Archit. News*, 35(1):63–64, Mar. 2007.

[39] IDA: About. http://www.hex-rays.com/products/ida/. Retrieved: March 21, 2012.

[40] T. Ince and J. K. Hollingsworth. Profile-driven selective program loading. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 62–73, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] Intel. Intel C and C++ Compilers | Intel Developer Zone. http://software.intel.com/en-us/c-compilers/. Retrieved: June, 2013.

[42] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[43] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, catamount. In *In Cray User Group*, pages 16–19, 2005.

[44] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[45] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy. Interprocedural static slicing of binary executables. 2003.

[46] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[47] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.

[48] M. Kotschenreuther, G. Rewoldt, and W. M. Tang. Comparison of initial value and eigenvalue codes for kinetic toroidal plasma instabilities. *Computer Physics Communications*, 88(2-3):128 – 140, 1995.

[49] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[50] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security and Privacy*, 9(3):49–51, May 2011.

[51] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software: Practice and Experience*, 24(2):197–218, 1994.

[52] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1995. ACM.

[53] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[54] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder. Reducing code size with echo instructions. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 84–94, New York, NY, USA, 2003. ACM.

[55] C. Lefurgy, E. Piccininni, and T. Mudge. Evaluation of a high performance code compression method. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 93–102, Washington, DC, USA, 1999. IEEE Computer Society.

[56] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[57] L. Li, J. E. Just, and R. Sekar. Address-space randomization for windows systems. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 329–338, Washington, DC, USA, 2006. IEEE Computer Society.

[58] Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 107–126, Berlin, Heidelberg, 2009. Springer-Verlag.

[59] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[60] S. Mansfield-Devine. Android malware and mitigations. *Network Security*, 2012(11):12 – 20, 2012.

[61] H. McGhan. SPEC CPU2006 benchmark suite. *Microprocessor Report*, 2006.

[62] Microsoft. .NET Framework. http://msdn.microsoft.com/en-us/vstudio/aa496123. Retrieved: June, 2013.

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

[64] Mozilla Corporation and Mozilla Foundation. Mozilla firefox web browser - free download. http://www.mozilla.org/en-US/firefox/new/. Retrieved: Feb 22, 2012.

[65] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV'03)*, 2003.

[66] Oracle. Oracle and Java | Technologies. http://www.oracle.com/us/technologies/java/overview/index.html. Retrieved: June, 2013.

[67] PaX Team. PaX address space layout randomization, Mar. 2003. http://pax.grsecurity.net/docs/aslr.txt. Retrieved: March 15, 2013.

[68] K. Pettis and R. C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, 1990.

[69] M. Prasad and T. Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2003.

[70] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*, AAAI'08, pages 798–804. AAAI Press, 2008.

[71] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 45–, Washington, DC, USA, 2002. IEEE Computer Society.

[72] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[73] sKyWIper Analysis Team. sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks. Technical report, Laboratory of Cryptography and System Security (CrySyS Lab), Budapest University of Technology and Economics, Budapest, Hungary, 2012. http://www.crysys.hu/skywiper/skywiper.pdf.

[74] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In R. Sekar and A. Pujari, editors, *Information Systems Security*, volume 5352 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin / Heidelberg, 2008.

[75] A. Srivastava, A. Edwards, and H. Vo. Vulcan binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[76] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[77] M. Szeredi. Ssh filesystem, 2005. http://fuse.sourceforge.net/sshfs.html. Retrieved: April 16, 2013.

[78] The Portland Group. PGI | Products | PGI Workstation. http://www.pgroup.com/products/pgiworkstation.htm. Retrieved: June, 2013.

[79] H. Theiling. Extracting safe and precise control flow from binaries. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, RTCSA '00, pages 23–, Washington, DC, USA, 2000. IEEE Computer Society.

[80] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[81] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, Athens, 12 2005. IEEE.

[82] L. Wang, C. Fang, B. Mao, and L. Xie. TMAC: Taint-based memory protection via access control. In *Proceedings of the 2009 Second International Conference on Dependability*, DEPEND '09, pages 19–27, Washington, DC, USA, 2009. IEEE Computer Society.

[83] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. 2012.

[84] Y. Xie, W. Wolf, and H. Lekatsas. Profile-driven selective code compression. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10462, Washington, DC, USA, 2003. IEEE Computer Society.

[85] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. *Reliable Distributed Systems, IEEE Symposium on*, 0:260, 2003.

[86] L. Zhang and C. Krintz. Profile-driven code unloading for resource-constrained jvms. In *PPPJ '04: Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, pages 83–90. Trinity College Dublin, 2004.

[87] A. Zmily and C. Kozyrakis. Simultaneously improving code size, performance, and energy in embedded processors. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 224–229, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.