Diversity of Generated Code in LLVM

Órla Ní Loinsigh
B.A.(Mod.) Computer Science
Final Year Project April 2015
Supervisor: Dr David Gregg

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

__________________________________________  _____________________________
Name                                                Date
Abstract

Currently, when software is distributed in binary form, every copy of that binary will be precisely identical for a given system. This may be termed a software monoculture. It is problematic for reasons of security, allowing the potential for millions of machines to be attacked successfully when a vulnerability is discovered in a single binary. This project explores the idea of compile-time security through diversity in generated binaries. This involves the creation of compiler techniques capable of generating diverse executables from a single source, that all implement that source correctly. Specifically, this entails the augmentation of LLVM, a production-grade compiler, with additional optimizer passes. The project will design and implement a variety of diversification techniques. It will test the binaries generated from this system on a variety of performance metrics. In addition, it will evaluate each technique for effectiveness in blocking a particular type of attack, known as stack-smashing. It will show that all of the techniques implemented are successful to some degree in preventing attack, and that one in particular, Function Re-Ordering, is extremely effective and with little overhead.
ACKNOWLEDGEMENTS

I wish to thank my project supervisor, Dr David Gregg, for his advice and feedback throughout the term.

I also wish to mention Andrew Anderson for his very helpful reading suggestions.

Lastly, I wish to thank Seamus, for always being so supportive of me.
# Contents

1 Introduction 1
   1.1 Motivation .................................................. 1
   1.2 Objectives .................................................. 1
   1.3 Technical Overview ......................................... 2
   1.4 Report Outline ............................................. 2

2 Background and Related Work 4
   2.1 Problem ....................................................... 4
      2.1.1 Software Monoculture .................................... 4
      2.1.2 Attack Types ........................................... 5
   2.2 Solution: Introducing Diversity .............................. 8
      2.2.1 Application Binary Randomization ......................... 8
      2.2.2 Related Work .......................................... 9
   2.3 Chapter Summary ............................................ 11

3 Diversification Strategies and Implementation 12
   3.1 Description .................................................. 12
   3.2 Platform and Target .......................................... 12
      3.2.1 LLVM Overview .......................................... 12
      3.2.2 LLVM Intermediate Representation ....................... 13
      3.2.3 LLVM Optimizer ....................................... 13
      3.2.4 Optimizer Passes ...................................... 13
   3.3 Pseudo-Randomness .......................................... 14
   3.4 Categorization of Strategies ................................ 14
   3.5 Module-Based Randomization ................................ 15
      3.5.1 Function Re-Ordering .................................. 15
   3.6 Function-Based Randomization ............................... 17
      3.6.1 Dummy Variable Insertion ............................... 18
      3.6.2 Basic Block Re-Ordering ............................... 19
   3.7 Chapter Summary ............................................ 20

4 Evaluation 21
   4.1 Testing Approaches in Related Work ......................... 21
   4.2 My Testing Approaches ..................................... 22
      4.2.1 Benchmarking .......................................... 22
4.2.2 Effectiveness .................................................. 25
4.3 Results ............................................................ 27
  4.3.1 Parameters ................................................... 27
  4.3.2 Dummy Variable Insertion ................................. 28
  4.3.3 Function Re-Ordering ...................................... 31
  4.3.4 Basic Block Re-Ordering .................................. 32
  4.3.5 Comparison .................................................. 34
4.4 Chapter Summary ............................................... 35

5 Further Work ..................................................... 36
  5.1 Amendments to Existing Randomization Techniques .... 36
  5.2 New Randomization Techniques ............................. 36
  5.3 Further Testing and Analysis Techniques .................. 37
    5.3.1 Benchmarking ............................................ 37
    5.3.2 Effectiveness .............................................. 37
  5.4 Chapter Summary ............................................. 38

6 Conclusions ....................................................... 39

A Sample PLT Attack ............................................... 40

B Auto-Generated Code for Effectiveness Testing ............... 43
  B.1 Examples of Auto-Generated Functions .................... 43
  B.2 Sample Main .................................................. 44

Bibliography ........................................................ 48

Associated code may be found on attached CD.
List of Figures

2.1 Analogy to illustrate software monoculture ........................................ 5
2.2 Analogy to illustrate software diversity ............................................. 9

3.1 Simplified version of the LLVM tool-chain .......................................... 12

4.1 Average effectiveness of Dummy Variable Insertion max-allocs=3 .......... 29
4.2 Average effectiveness of Dummy Variable Insertion max-allocs=5 .......... 29
4.3 Effectiveness of Dummy Variable Insertion max-allocs=3 ................. 30
4.4 Effectiveness of Dummy Variable Insertion max-allocs=5 ................. 30
4.5 Average effectiveness of Function Re-Ordering ................................... 32
4.6 Effectiveness Function Re-Ordering .................................................. 32
4.7 Average effectiveness of Basic Block Re-Ordering .............................. 33
4.8 Effectiveness Basic Block Re-Ordering ............................................. 34
4.9 Comparative retired instructions ...................................................... 34
4.10 Comparative average effectiveness ................................................... 35

A.1 State before calls to memcpy ............................................................ 42
A.2 State after calls to memcpy ............................................................... 42

Note: unless otherwise stated, all graphs and images are my own.
List of Tables

4.1 Benchmarking results for Dummy Variable Insertion. ................. 28
4.2 Benchmarking results for Function Re-Ordering. ................. 31
4.3 Benchmarking results for Basic Block Re-Ordering. ................. 33
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization.</td>
</tr>
<tr>
<td>BBR</td>
<td>Basic Block Re-Ordering.</td>
</tr>
<tr>
<td>DVI</td>
<td>Dummy Variable Insertion.</td>
</tr>
<tr>
<td>FR</td>
<td>Function Re-Ordering.</td>
</tr>
<tr>
<td>GOT</td>
<td>Global Offset Table.</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation.</td>
</tr>
<tr>
<td>LLVM</td>
<td>Acronym now defunct (originally Low Level Virtual Machine).</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System.</td>
</tr>
<tr>
<td>PLT</td>
<td>Procedure Linkage Table.</td>
</tr>
<tr>
<td>ROP</td>
<td>Return-Oriented Programming.</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment.</td>
</tr>
</tbody>
</table>
Glossary of Terms

**Address Space Layout Randomization, ASLR:** A security protection offered by modern operating systems to randomize library code base section addresses.

**Basic block:** A maximal sequence of instructions with a single entry point and a single exit point.

**Buffer overrun, buffer overflow:** A vulnerability where it is possible to run over the end of an allocated area of memory such as an array.

**GOT.PLT:** A structure used to facilitate dynamic linking of library functions.

**Intermediate Representation, IR:** An intermediary language used by compilers internally, independent of source language and target assembly language.

**Knuth shuffle, Fisher-Yates shuffle:** A shuffling algorithm that produces unbiased orderings.

**libc:** C standard library.

**LLVM:** A production-grade compiler tool-chain.

**φ-node:** An abstract function inserted when assignment may occur along two or more control-flow paths in SSA form.

**Retired instructions:** Instructions that are entirely completed, i.e. whose effects are written back.

**Return-Oriented Programming, ROP:** An attack type where addresses of executable sequences are arranged such that only their returns are used.

**Software gadget:** A routine or part of a routine in executable memory that may be used in an ROP attack.

**Stack frame:** An area of memory on the call stack encapsulating all data pertaining to a particular function call.

**Static Single Assignment, SSA:** A representation where each variable may be assigned to only once.
Chapter 1

Introduction

1.1 Motivation

The motivation behind this project is the importance of digital security. Security may be breached for a variety of reasons, but the focus in this project is on breaches arising from vulnerable code.

A typical result of a breach in security is a data breach, where sensitive data (e.g. financial or medical records) are copied, deleted, or released publicly following a successful exploit. A 2014 review by the Ponemon Institute and IBM [Pon14] notes that the cost of an average data breach is typically in the millions, and is rising year on year in almost every country studied. Some of this, such as the cost of rebuilding resources and discounts for those affected, is direct and quantifiable. The remainder is difficult to measure, and rests on indirect factors like the loss of customers and future business that may result from even a relatively minor breach in security.

This establishes the ongoing relevance of securing vulnerable code.

1.2 Objectives

The overall objective of this project is to enhance binary security. In this case, this does not mean security of individual binaries, but rather, security of the community of binaries as a whole.

This is to be achieved by investigating and implementing compile-time diversity. There are many different ways that a given source may be implemented in assembly. The idea behind my project is to exploit this property to produce binaries that are subtly different (enough to
block an attack in some cases) while still implementing the source correctly. The aim is to end up with a compiler tool-chain that is capable of generating diverse binaries.

The intention is to investigate techniques for diversification, and select a few to implement. The focus of this project is real-world attacks, and so the implementation will take place within a real-world compiler (LLVM).

Lastly, I will evaluate the implemented techniques on a variety of metrics, such as effectiveness in blocking attacks and the potential for overhead penalties.

1.3 Technical Overview

Implementation of this project will involve augmenting LLVM, which is a production-grade compiler. In particular, I will be writing passes for the LLVM optimizer, opt. Most of the discussion on implementation in this report will be with reference to LLVM. However, to outline some principles more clearly, some mention is made of methods used in other compilers (particularly GCC).

The passes written for this project are relatively stand-alone. These passes need to be connected into the LLVM infrastructure for the augmented tool-chain to be complete. This is prototyped in this project using a series of scripts, and incorporated into testing. Testing is considered important in this project, as the output of this project is not so much the compiled passes, as the binaries produced from the amended compiler tool-chain. Therefore, a large amount of the coding for this project involves creating test frameworks that demonstrate certain properties about the generated binaries.

1.4 Report Outline

Chapter 2 will discuss the background to this project, including the nature of software monoculture and the sorts of attacks that are exacerbated by it. It will outline the idea of diversity as a solution, and discuss some of the related work in the area of binary randomization.

Chapter 3 will describe my diversification strategies, their design, and how they are implemented in LLVM.

Chapter 4 will discuss evaluation of the implementation. This chapter includes both a detailed description of the testing methods constructed, and the results achieved when comparing my augmented LLVM system to the default system.
Chapter 5 will outline further work in the area of compile-time application binary randomization.

Chapter 6, lastly, will summarize my project and its findings.
Chapter 2

Background and Related Work

2.1 Problem

2.1.1 Software Monoculture

A monoculture is a community in which all individuals are identical. It is typically relevant to biological populations and agriculture; there, the problem of monoculture is that all individuals are vulnerable to the same pathogens.

Software monoculture is the analogous problem where identical copies of the same binary are vulnerable to the exact same attack \([\text{JSH}^{+11}]\)[\(\text{HNL}^{+13}\)]. After identifying and targeting a vulnerability on one copy of the binary, an attacker will be able to replicate this on all other deployed copies of the binary. For example, an attacker may disassemble a binary and determine an input that exploits some underlying vulnerability and gives the user control of the host machine. Because all binaries are identical, that same input will work on all other systems where it is installed. This leaves potentially millions of machines open to attack at no additional cost to the attacker.

As an analogy, imagine a town in which all houses had the same lock. Should a burglar be able to obtain or fashion a copy of a key that opens one of the locks, then because all locks are identical, they will be able to access all of the houses. In this analogy, the house is a binary, the lock is the binary’s potential vulnerability, and the key is some input or payload that may be thrown at the binary to effect an attack.

A classic example of such a community is x86 binaries running on a particular Windows distribution. Identical copies of so many programs exist that, when a flaw is found with one copy, all machines with a copy are open to attack. It is for this reason that that particular platform arrangement sees such a high proportion of targeted attacks \([\text{WMHL12b}]\).
Chapter 2. Background

FIGURE 2.1: Analogy to illustrate software monoculture

However, the same flaw exists in any system that implements mass identical binary distribution. An example of this is the recent GHOST vulnerability that appeared in specific functions in the GNU C library (glibc) [Qua15]. This vulnerability features a heap-based buffer-overflow with the potential to allow the attacker to execute arbitrary code. It existed in every version of glibc from 2.2 (November 2000) to 2.17 (December 2012), and the vulnerability was not officially understood as a security threat until January 2015. Most systems at that time still had versions of glibc from before 2012, meaning literally millions of machines were affected.

2.1.2 Attack Types

There are several attack types that are relevant to the problem of software monoculture, including buffer-overrun attacks, memory allocator manipulation, program tampering, etc. [LHBF14]. Some of these attack types and approaches can (and have) been mitigated by operating-system-level protections. Others are more difficult to tackle at the OS level and require a different approach.

2.1.2.1 Stack-Smashing

The focus of this project will be a type of buffer-overrun attack called stack-smashing. Here, the contents of the call stack are manipulated through a buffer overrun, with the purpose of manipulating program control flow. The aim is generally to gain control of the host machine, leaving it open to arbitrary malicious actions.

A well-known article by Aleph One from 1996 [Ale96] describes the principles of a stack-smashing attack. It demonstrates how easy it is in theory to smash the call stack, using little more than a text editor and a disassembler. While the exact techniques used in it are generally not technically possible anymore (mainly due to better protection by operating systems), the principle of smashing the stack as a means of maliciously targeting a machine is no different today [jip12].
2.1.2.2 Attack Approaches

There are different approaches to gaining control of a machine through stack-smashing. Due to various defence mechanisms that have been developed over time, there has been a natural progression in the preferred type of attack. Nevertheless, they are still worth noting as part of the background to this project.

Code Injection  In a code injection attack, the attacker writes instructions into somewhere in memory and then redirects flow control so that they are executed. An example of this would be an executable stack. A stack buffer overrun causes the contents of the stack to be replaced with executable code. This code can then be executed directly.

Code injection has been largely defeated by an OS protection known as Data Execution Prevention, also known as W⊕X. Under this scheme, every memory page may be either writable or executable, but not both. In particular, this means that the stack is not typically executable. This makes it impossible to directly execute code that has just been inserted.

Return-to-libc  Where direct arbitrary code injection is not possible, existing code may be manipulated instead. In general, this means identifying particular routines within existing code and manipulating the stack through buffer overruns to cause these to be run in a useful sequence (perhaps leading them to give an attacker control of the target machine). Note that this is not solved by W⊕X, as the code being used to attack the machine already exists in the executable area of memory.

Return-to-libc attacks make use of library code to this end. The stack is manipulated so that the return address now points to some routine in libc. Due to its ubiquitous nature, libc will nearly always be linked, and it contains a great many useful routines. A typical approach is to call a library function like system() or exec1(), which run commands in the host’s shell; commonly, the command to be run will be one that spawns a new shell. Ideally, the spawned shell will have the permissions not of the user running the program, but of the owner of the program (potentially an administrator or root).

Return-to-libc attacks are partially prevented by another OS protection mechanism, called Address Space Layout Randomization (ASLR). ASLR is the randomization of library code base sections, such as those that make up libc. Every time an application is started, the library functions it links to are placed in different random locations in memory. In theory, this prevents the sorts of attacks that depend on knowing the run-time addresses of libc functions. While ASLR is effective in dealing with several types of attack, it has a variety of limitations.
Firstly, it is only really effective on 64-bit platforms. On 32-bit platforms, only a limited number of bits are available for randomization, and this can be overcome by brute-force. One study [SPP+04] demonstrates a particular implementation of ASLR on a 32-bit system being defeated in an average of 216 seconds. This problem extends to applications running in 32-bit mode on a 64-bit platform. On 64-bit systems, typically upwards of 40 bits are available for randomization, which should be sufficient protection against brute-force attacks.

Secondly, ASLR is typically not applied to application binaries. Return-oriented programming attacks (see below) are capable of finding the routines they need within application binaries rather than library code [WMHL12a], so these can still circumvent ASLR.

Lastly, ASLR arguably does not randomize with enough granularity [WMHL12a][SPP+04]. If increasing granularity in space, randomization would preferably be on a basic-block level, rather than at section base addresses only as ASLR currently does.

Return-to-PLT As noted above, the addresses of library functions cannot be known at compile-time. Instead, they must be resolved by the dynamic linker. A simplified version of this procedure is as follows. To facilitate dynamic linking, a table called the Global Offset Table/Procedure Linkage Table (GOT.PLT) is allocated at compile-time. The table consists of jump slots, one for each of the library functions the application uses. At run-time, the slots will be filled in by the dynamic linker. When a library function is called for the first time, the dynamic linker resolves its real address; every successive call to that library function can then be found by simple look-up in the GOT.PLT [Lev99].

A return-to-PLT attack manipulates this table so as to disrupt flow control. Slots of the GOT.PLT are themselves writable, and what they point at may be executable. Therefore, if through some program vulnerability we manage to write to the GOT.PLT, we may fool the program into believing it is executing a library function when in fact it is executing something of our choosing. A worked example of a PLT-related attack is found in Appendix A. It is based on a 2012 article from soldierx.com [jip12]. For the variation given, some names have been changed and some elements rearranged, but it is effectively the same example. My project employs this code to demonstrate the effectiveness of the randomization strategies implemented (see Chapter 4).

Return-Oriented Programming In return-oriented programming (ROP), routines (termed gadgets) that will be used to attack the machine are found in the application binary itself. Gadgets end in a return instruction. In ROP attacks, gadget addresses are arranged on the stack in such a way that only their returns (not the corresponding calls) are used. The stack pointer effectively becomes the instruction pointer [CLBF13].
In theory, such an attack may be detected and overcome by checking for odd patterns of returns (specifically, frequent returns [CLBF13]). In particular, finding large numbers of mispredicted returns (those without corresponding calls) would be a red flag. However, the ROP attack technique may be generalized to Jump-Oriented Programming, where the same effect is achieved through direct or indirect jump instructions.

2.2 Solution: Introducing Diversity

2.2.1 Application Binary Randomization

The problems of return-to-PLT and ROP attacks remain unresolved. As described above, the defensive approaches implemented on a large scale so far have generally been at the OS level. These approaches are certainly effective to an extent, but are not particularly effective at dealing with PLT and ROP-type attacks.

The approach to be taken in this project will be to diversify application binaries, varying things like memory addresses and stack frame layout. The intention is to severely limit the damage that may arise from a software monoculture. Under a monoculture, if a vulnerability is discovered, then an attacker’s successful exploit may then be replicated on millions of other machines with no further effort. In a diverse culture, while some copies might still be vulnerable, the attacker would have to begin the attack anew with each differing binary. Thus, the idea behind diversification is not so much making attack impossible as making attack not worthwhile [Coh92].

There is an infinite number of assembly arrangements that correctly implement a given source. Some of the choices made in compilation are for a specific reason, for example to exploit code locality or allocate registers such that spills to memory are minimized. Other choices may be arbitrary. This project will explore exploiting this property in order to produce different binaries. The differences in these binaries will not be arbitrary, but rather, carefully chosen from a small pool of transformations. The binaries should still implement their source correctly, but may be structurally different enough to no longer be vulnerable to a specific attack.

To return to our earlier analogy, if the houses all have different locks, it is now still possible that the burglar may gain access to one, should they find the appropriate key. However, unlike before, this will not mean they automatically have access to all of the town, merely to a greatly-reduced subset of it.
2.2.2 Related Work

This section will discuss selected approaches that have been proposed in the area of application binary randomization. Research in the field can be found from the early 1990s onwards. However, the focus of evaluation has shifted over the years to accommodate updates in hardware capabilities and computing practices.

Prior to around 2008, there was little in the way of implementation or even discussion of compile-time randomization. Security in the intervening years appears to have focused on OS-level protections instead. However, in recent years, there has been a resurgence of interest in the area.

The discussion of related work in this chapter serves to give a simple overview of some of the schemes that have been implemented. For a discussion of their methods of evaluation, see Chapter 4.

The focus of my project is compile-time randomization, but both compile-time and run-time approaches are included here for completeness.

2.2.2.1 Compile-Time

An early mention of the potential of diversity as a defensive technique is found in a 1992 paper by Frederick Cohen [Coh92]. The paper details the value of coming up with a system of defences (one that is not itself prohibitively costly) that would render attacks no longer worthwhile (even if they might ultimately succeed). The author notes that automation would be needed to come up with something different for every machine in the world. A variety of diversification methods (termed “evolutions”) are suggested and implemented. Evaluation focuses on metrics that are no longer particularly relevant today, such as compilation time and ease of distribution.

There are certainly still some issues around the distribution of diversified binaries that remain to be fully solved, for example around the area of software updates; this is noted in a later section. What is referred to here is the issue of physical distribution in the age before widespread Internet access.
Another early work by Forrest et al. (1997) [FSA97] gives a broad overview of why one would want to diversify, and of the various possible methods. Some techniques in the area of stack randomization are explored on the then-current version of Linux. Some preliminary results are given, but the paper notes the need to evaluate other methods.

The approach described by Bhatkar and Sekar (2008) [BS08] sets out to achieve diversity by randomizing how data is represented in a program. Program variables are randomized by masking them on assignment, and then unmasking them on use. It is intended to guard against attacks based on memory corruption, such as heap overflows and stack buffer overruns. The implementation translates from C code to C code.

An assortment of techniques for securing OS code are discussed by Giuffrida et al. (2012) [GKT12]. In particular, it focuses on address space randomization. The idea is that every user would receive the same binary of their OS. Randomization is at a combination of compile-time and link-time, with some background process tasked with linking in new variants of OS components.

Homescu et al. (2013) [HNL+13] propose a randomized NOP-insertion scheme. Diversity is achieved by inserting garbage code in the form of NOP instructions unpredictably throughout the code. The insertion of NOPs causes the relative offsets of other instructions in the code to be altered. Attacks that rely heavily on these offsets remaining constant may therefore be blocked by this technique. The approach of Homescu et al. profiles the code as “hot” and “cold”. Its primary insight is that NOP insertion is more efficient when it is restricted in “hot” regions. Compared to naïve insertion, such an approach delivers a similar result in terms of security benefits, for a greatly reduced execution time overhead. After tuning to determine the optimal frequency of NOP insertion, they report performance overhead as low as 1%.

2.2.2.2 Retrofitting Existing Binaries

Wartell et al. (2012) [WMHL12a] explore the possibility of retro-fitting existing x86 binaries with randomized instruction addresses: a useful approach in the absence of source code, debugging symbols, etc. Moreover, the input binary is transformed into a self-randomizing binary, one that is capable of re-randomizing itself every time it is loaded anew.

A related approach is detailed in another paper by Wartell et al. from the same year [WMHL12b]. This paper notes the current problems involved in rewriting existing binaries, specifically, that some additional information (source code, debugging symbols from a specific compiler, etc.) may be required. This particular approach, which focuses specifically on Windows x86 binaries (due to their popularity among attackers), proposes a rewriting system where such information is assumed to be unavailable.
2.2.2.3 Countermeasure Insertion

The more aggressive approach argued for by Crane et al. (2013) [CLBF13] is worth mentioning. The proposal here is to actively discourage attackers from attempting brute-force attacks in particular. The authors assert that current approaches are too passive, in that they are only ever reactive to an attack strategy. As a result, the attacker is always at an advantage, as they have little to lose by attempting an attack.

The general approach is to insert booby-trap “hooks” into a binary generated by diversification, and, rather than just leading the program to fail, to transfer control to a defence sequence. They distinguish compile-time insertion, which would allow greater flexibility to the defender, from load-time insertion, which would allow the same binary to be distributed and then diversified at load-time. The proposed defence sequences range from recovery-handling (such as undoing the changes made by the attacker), to garnering information about the attacker (with possible uses in profiling), to outright counterattack (noted for its questionable safety and legality).

2.3 Chapter Summary

This chapter has described the background to the problem of particular types of targeted attack, and how that problem is exacerbated by the nature of software monoculture. It has noted diversity as a possible solution to this problem, and summarized some of the related work, past and present, in that area. The next chapter will discuss the strategies for diversification explored by my project.
Chapter 3

Diversification Strategies and Implementation

3.1 Description

This chapter will describe the randomization techniques explored in this project. It will discuss which strategies were investigated, why they were chosen, and how they may be categorized. It will also describe how they are implemented in the LLVM system.

3.2 Platform and Target

As proof of concept, a variety of randomization techniques have been prototyped. This section describes the infrastructure used to achieve this, LLVM.

3.2.1 LLVM Overview

LLVM is a compiler infrastructure. It provides a framework for a variety of compiler tools, with modular components for compilation, assembly, optimization, etc.

![Figure 3.1: Simplified version of the LLVM tool-chain](image)

The target version for this project is LLVM release 3.5.0 (September 2014).
3.2.2 LLVM Intermediate Representation

Intermediate Representation (IR) is an intermediary language used by compilers internally. Different compilers use different types of IR [GCC][LLV14a].

In LLVM, source code is taken in by the compiler front-end in some high-level language (e.g. C), parsed and type-checked, and converted into LLVM IR. Transformations may be performed on the IR to optimize it; examples are constant propagation and dead code elimination. IR is then compiled into its target assembly language (e.g. x86, ARM). Assembly code is assembled into object files, and then linked to form an executable.

IR in LLVM is in Static Single Assignment (SSA) form. SSA is a representation in which every variable is assigned to only once. When there is the possibility that a variable may be assigned a different value depending on which path is taken to reach that assignment, a $\phi$-function must be inserted [Muc97]. In LLVM, this is represented by a $\phi$-node instruction.

Operations on IR in LLVM are separate from operations at the front-end (lexing and type-checking) and operations at the back-end (code generation). In this way, optimizing operations that involve translating IR to IR are both source- and target-independent. Examples of such operations are passes written for the LLVM optimizer.

3.2.3 LLVM Optimizer

The LLVM optimizer, opt, operates by performing passes for analysis and transformation on LLVM Intermediate Representation (IR). When analysing, opt takes in an LLVM IR file and outputs the results of analyses. When transforming, it loads the requested modules (or passes), takes in an LLVM IR file, performs modifications, and outputs another LLVM IR file [LLV14b].

My implementation takes the form of IR-to-IR optimizer passes. These passes are compiled individually into shared objects, and then linked dynamically when loaded by opt [LLV14c]. This allows passes to be written and compiled quickly, without the need to re-compile the entirety of LLVM every time.

3.2.4 Optimizer Passes

There are several different pass types; the ones most relevant to this project are as follows [LLV14c]:

...
• **ModulePass**: a pass that is run on every module (or file) in the program; no other modules can be accessed, i.e. when operating on one module, we cannot access the functions etc. of another.

• **FunctionPass**: a pass that is run on every function; when operating on one function, we cannot access the basic blocks etc. of another.

• **BasicBlockPass**: a pass that is run on every basic block; when operating on one basic block, we cannot access the instructions etc. of another.

The optimizer provides a framework for running user-defined passes. In addition, command-line parameters may be passed to user-defined passes. My implementation makes use of this framework.

Note that, while LLVM is notionally both source- and target-independent, results here will be discussed in terms of the quality of the x86 code ultimately generated. The reason for this is simply the availability of x86 machines.

### 3.3 Pseudo-Randomness

It should be noted that where this report talks of “random”, what is actually meant is “psuedo-random”. Seeds for random-number generators are taken from parameters passed by command-line (rather than taken from, say, system time). The output of the randomizing compiler will be deterministic for a given seed. There are several reasons why this is desirable.

One reason is verifiability. In order for properties of binaries to be verified, they must be reproducible. If we were to get truly random executables every time, it would be impossible to prove that they were correct.

Another reason stems from an issue raised by Franz (2010) [Fra10]. Randomizing applications poses a problem when it comes to software updates. However, if each version is generated using a specific seed, then this seed may also be used to create or retrieve updates appropriate to that version.

### 3.4 Categorization of Strategies

Randomization techniques can be categorized as follows.

• Whole-program.
Module-based, i.e. within a module (or file).

• Function-based, i.e. within a function.

• Basic-block-based, i.e. within a basic block.

Randomization techniques at whole-program level take the program as a whole; in other words, they take place at or after link-time. Because the focus of my project is compile-time diversification, no whole-program techniques were investigated.

Basic-block-based techniques restrict randomization to within a basic block. Here, we can manipulate the instructions of a basic block, but cannot access the instructions of another basic block. A basic block is defined as a maximal sequence of instructions with a single entry point and a single exit point; in other words, flow control cannot jump into or out of the interior of a basic block. Basic blocks typically (but not always) begin with a label, and end with an instruction like a jump or return. Thus, functions consist of basic blocks arranged in a control flow graph, which is a graph representation of the transfer of control flow through the function.

Due to time constraints, no techniques at basic-block level have been implemented as part of this project. These may be explored as further work. As such, this project focused on module-based and function-based techniques.

3.5 Module-Based Randomization

Here, we restrict randomization to within a program module (or file). We can manipulate the functions and data of a module, without reference to the functions of another module. This project investigates one module-based randomization technique, Function Re-Ordering.

3.5.1 Function Re-Ordering

3.5.1.1 Description

Function re-ordering is a strategy for randomizing code.

This involves the re-ordering of functions within a module. Functions can be placed in any order within a program [LHBF14], without causing any functional difference in its operation. In unoptimized compilation, functions will be outputted in the order they appear in in the source. With optimizations enabled, compilers may alter the order of functions to improve performance [GCCb][Mic]. This is typically done by profiling the code as “hot” and
"cold", and grouping functions that call each other together in order to exploit code locality. Optimized or not, output will be deterministic.

My function re-ordering pass re-arranges the functions of a module in random order. This loses the benefit of any other ordering that might exploit code locality, but should make the location of program symbols more difficult for an attacker to predict. My pass must be run in place of any other function re-ordering passes; at present, LLVM does not have a standard function re-ordering pass.

### 3.5.1.2 Implementation

In theory, FR could be implemented at a variety of different stages in the compiler tool-chain. Among them are:

- At source level: re-ordering functions by transforming the source file to another source file.
- At intermediate representation (IR) level: re-ordering functions by transforming IR to IR.
- At assembly level: re-ordering functions by transforming target assembly to target assembly, or while compiling IR to target assembly.

Re-ordering functions at source level would require a complicated level of analysis; functions would have to be forward-declared appropriately, and careful consideration would have to be given to the placement of variable declarations. Failure to do so may lead to compilation failing at the compiler front-end.

Re-ordering at assembly level is also problematic; the code-generation stages that generated the assembly will have determined whether a particular function call is direct or indirect, and re-ordering them at random at this stage would require these decisions to be re-evaluated.

In contrast to the above, an IR-to-IR transformation pass has no such complications. When working at IR level, the front-end has already ensured the correctness of the function calls being made. In addition, code generation has yet to take place, so no further overhead is added there. For these reasons, IR-to-IR transformation was the obvious choice.

FR is implemented as a ModulePass in LLVM, meaning it is done on a whole-module basis. The pass takes a single parameter, `-rnd-seed`, a value to seed the random number generator.
3.5.1.3 Algorithm

The algorithm for function re-ordering is somewhat trivial. The underlying algorithm is an adaptation of a basic Knuth shuffle. A Knuth shuffle (or Fisher-Yates shuffle) produces unbiased orderings; we are not more likely to get some orderings than others. Given a function `generate_uniform_random(p, q)`, which generates uniformly-distributed random integers between `p` and `q`, inclusive, the algorithm to shuffle a list `L` with length `len` in-place using a Knuth shuffle is:

```python
for i in 1..len do:
    index := generate_uniform_random(i, len)
    swap(L[i], L[index])
```

3.5.1.4 In LLVM

In LLVM, each module stores its functions in list form. Re-ordering is achieved with a simple list shuffle. In my implementation, shuffling is achieved in two passes. The first removes functions from the module and places them in a temporary list in order. The second iterates over the temporary list, returning functions to the module at random.

```python
rng.seed(rnd_seed)
l = list<function> tmp

for module in program do:
    for func in module.functions do:
        module.remove(func)
        tmp.add(func)

for func in tmp do:
    index := generate_uniform_random(0, tmp.size()-1)
    module.functions.add(tmp[index])
    tmp.remove_at(index)
```

3.6 Function-Based Randomization

Here, we restrict randomization to within a function. We can manipulate the basic blocks of a function, without reference to the blocks of another function. This project investigates the following function-based randomization techniques:

1. Dummy Variable Insertion.
2. Basic Block Re-Ordering.
3.6.1 Dummy Variable Insertion

3.6.1.1 Description

Dummy variable insertion (DVI) is a strategy for randomizing stack frame layout. On the call stack, a stack frame encapsulates all data pertaining to a particular function call, such as parameters passed to it, local variables, the return address, etc.

In most architectures, local variables are allocated on the stack. A dummy variable is one that would not be used by the program; it is neither assigned nor referenced. When a random number of dummy variables is inserted, structure of a stack frame is varied without changing program meaning.

In theory, DVI could be implemented at basic block rather than function level. The reason this is not done is that some basic blocks will be inside of loops, sometimes deeply-nested ones. Inserting useless variables in such locations would be a misuse of resources.

3.6.1.2 Implementation

DVI is implemented as a FunctionPass in LLVM; in other words, dummy variable insertion is done on a per-function basis. The pass takes two parameters:

- `-rnd-seed`: a value to seed the random number generator.
- `-max-allocs`: a value indicating the maximum number of variables that may be inserted.

It is necessary to bound the number of variables that may be inserted for practical reasons; without such a bound, we could potentially have millions of variables being allocated. It was decided to leave this bounding choice up to the user.

3.6.1.3 Algorithm

The algorithm is straightforward:

```plaintext
rng.seed(rnd_seed)

foreach function in module do:
  entry := function.entry_block
  quantity := generate_uniform_random(0, max_allocs)
  for i in 0..quantity do
    instruction := new Instruction("alloc.local")
    entry.insert(instruction)
```
3.6.1.4 In LLVM

There is an instruction in LLVM to allocate space for a local variable, `alloca`. The `alloca` instruction specifies size; in my implementation, every dummy inserted is a 32-bit integer.

The above algorithm is adapted so that, for every function, between 0 and `max_allocs` `alloca` instructions are inserted. They are placed immediately before the first instruction that is not a $\phi$-node instruction in the function’s entry block. The entry block is the first node in the function’s control flow graph.

3.6.2 Basic Block Re-Ordering

3.6.2.1 Description

Basic block re-ordering is a strategy for randomizing code.

This involves the re-ordering of basic blocks within a function. While all linkages between a basic block and its successors/predecessors must be maintained, the blocks themselves may be arranged in any order within the function. Like function re-ordering, a non-randomized basic block re-ordering pass is sometimes run by compilers [GCCb] to optimize generated code. In the case of GCC, this is done “in order to reduce number [sic] of branches taken and improve code locality”, and is run as part of the -O2 optimization level. At present, LLVM does not run a basic block re-ordering pass. If it were to in the future, then if my basic block re-ordering pass were run, it would have to be in place of such a pass.

3.6.2.2 Implementation

For reasons similar to those stated for Function Re-Ordering above, re-ordering basic blocks at IR level was by far the most sensible choice.

BBR is implemented as a `FunctionPass` in LLVM, as basic blocks are re-ordered within each function. Like FR, this pass takes a single parameter, `-rnd-seed`, to seed the random number generator.

3.6.2.3 Algorithm

The underlying algorithm for BBR is the same Knuth shuffle used for FR (see above).
3.6.2.4 In LLVM

Again, the implementation for BBR is almost the same as that for FR. There is one additional restriction with BBR: in LLVM, a function’s entry block may not be moved.

\[
\begin{align*}
\text{rng.seed(rnd_seed)} \\
\text{list<bbblock> tmp} \\
\text{foreach function in module do:} \\
\text{foreach bbblock in function.blocks [2..end] do:} \\
\text{function.remove(bbblock)} \\
\text{tmp.add(bbblock)} \\
\text{foreach bbblock in tmp do:} \\
\text{index := generate_uniform_random(0, tmp.size()-1)} \\
\text{function.blocks.add(tmp[index])} \\
\text{tmp.remove_at(index)}
\end{align*}
\]

3.7 Chapter Summary

This chapter has described which randomization strategies were chosen and why. It has discussed their implementation: the LLVM platform they were integrated into, the reasons for specifying pseudo-random rather than completely random seeding, and the particulars of the passes themselves. The next chapter will discuss how this system was evaluated.
Chapter 4

Evaluation

4.1 Testing Approaches in Related Work

This section discusses how testing has been carried out in other work relating to randomization, which influenced the testing methods chosen for my project.

Over the years, several randomization techniques have been implemented, using various strategies on various platforms. There is no uniform approach to testing or evaluation. The earliest techniques \[Coh92\] are concerned with the problems of compilation time, and how programs are to be distributed in bulk in the time before access to the Internet was common. Most of the later approaches discuss execution time \[FSA97][BSD05][BS08][GKT12][HNL+13]\; some also analyse memory usage \[FSA97][GKT12\]. In other words, where evaluation is concerned, benchmarking has been relatively well-covered.

What is lacking from some of the related work is an experimental approach to determining effectiveness. In some cases, such an assessment is entirely absent or unreported (sometimes stated as a binary condition of success, rather than giving any measure of effectiveness). In other cases, effectiveness is evaluated theoretically rather than practically. Bhatkar et al. (2005)[BSD05\] argue for an analytical approach over an experimental one, citing limitations in contemporary attacks. Bhatkar and Sekar (2008)[BS08\] state this again, citing the fragility of attacks and the time-consuming nature of amending them as reasons for preferring an analytical approach. Giuffrida et al. (2012)[GKT12\] argues the same, citing the above papers as reasoning.

There are a few experimental approaches, such as Homescu et al. (2013)[HNL+13\], where effectiveness is inferred from the movement of software gadgets. While relevant, the results reported in this paper do not cover the same ground as my project; the technique being evaluated is randomized NOP-insertion (a technique not implemented here), and the attack being
investigated is ROP (rather than PLT). In general, there are few reports that describe effectiveness in terms of anything real-world. This was a problem I wished to address with my project.

4.2 My Testing Approaches

Two testing mechanisms have been constructed as part of this project. One test mechanism benchmarks large-scale programs on a variety of metrics, while the other assesses variably-sized programs specifically for effectiveness in blocking stack-smashing attacks.

Tests were run on a variety of Linux distributions (Ubuntu and Debian), and so the frameworks and tooling described relate specifically to such a system.

4.2.1 Benchmarking

4.2.1.1 Description

This testing mechanism investigates the behaviour of large-scale programs compiled with the three randomization techniques. It is desirable to determine the effect that randomizing generated code will have on program operation. It is also desirable to get results as accurate as possible, and for them to reflect the performance of large, real-world programs.

Therefore, the requirements for this testing mechanism were as follows:

- The selection of appropriate metrics.
- A large input real-world code-base.
- A framework to run tests in bulk.

4.2.1.2 Metrics

The following metrics were considered useful for benchmarking:

- Generated binary size.
- Execution time.
- Memory usage.
To this end, a variety of tests were designed and scripted. Determining generated binary size can be done without running the generated executable. The remaining metrics are assessed with respect to a particular given task. More detailed descriptions are given below.

**Generated Binary Size**  Binary size can be determined simply with the system tool `du`.

**Execution Time**  Determining execution time proved to be problematic, due to the unpredictable nature of system load etc. on a given run while using a machine with multiple concurrent users. It was considered more reliable to use retired instructions as a proxy for estimating performance. Retired instructions are those that are completed and whose effects are written back (as opposed to other instructions, whose effects may be discarded because of, say, a branch misprediction).

Retired instructions may be measured with a tool called `perf`. This tool reports performance analysis on a variety of software and hardware events. While (depending on the installation) it may not have a counter dedicated to counting retired instructions, it provides the facility to access hardware counters directly. As a result, this particular test may only be run on non-virtualized machines.

Hardware counters are accessed by `perf` by means of passing the appropriate hardware code. As such, this test is also hardware-specific (or at least, specific to a hardware family). My tests were all run on machines with Intel Xeon chips, and therefore my testing script uses the code for retired instructions appropriate to that target `[Int]`.

It is necessary to take an average of several readings because of the differences that may arise among different runs to do with external factors such as machine load. The higher the value of `readings_per_run` (see below) is, hopefully the more accurate a representation of true retired instructions we will get.

To further compensate for hardware factors, when taking the average, we first run the test once and discard its result. This is so that, on the second and subsequent runs, the code is more likely to have been read into a cache and so will be more consistent. When testing this initially, it was found that failing to account for this meant that the first reading was around 20% higher than all subsequent readings, which was throwing the result significantly.

**Memory Usage**  Heap memory is measured using the tool `valgrind`. This provides a report on total heap usage for a given program run.

Heap usage is not expected to be different among randomized and non-randomized versions, as this should be deterministic for a given task. This is shown to be the case.
No test has been devised to measure stack memory usage. This may be implemented as further work (see Chapter 5).

4.2.1.3 Code-Base

The code-base I chose to work with was the source for sqlite3, a database program. What made this an attractive option is that the source is distributed as a single large file, such that all necessary functions are self-contained. The file consists of around 150,000 lines of code and 1,800 functions. I split the file into several constituents:

1. A ‘header’, consisting of all global declarations, struct declarations, preprocessor directives, etc.
2. Forward declarations: these may be auto-generated from the original source using a tool called cproto.
3. Functions: all program functions, each in a separate file (several, such as those that were Windows-specific, were pruned).

4.2.1.4 Procedure

We define the following parameters:

1. total_versions: the number of base versions to generate.
2. runs_per_technique: the number of times we are to run each randomization technique on each base version.
3. readings_per_run: the number of readings to average across.

The procedure to construct and run the tests is then as follows.

generate array of seeds of length runs_per_technique

for i in 1..total_versions do:
generate base source version
  (concatenate header, forward declarations, and functions in a randomized order)
  compile the C source to LLVM IR
  optimize the IR to -O2
  compile the IR to target assembly
  assemble and link the assembly into an executable
run each of the test scripts on the generated base executable
for j in 1..runs_per_technique do:
  for technique in randomization_techniques do:
    randomize the IR with technique using seeds[j]
4.2.2 Effectiveness

4.2.2.1 Description

The aim of this testing mechanism is to investigate the effectiveness of each strategy in blocking stack-smashing attacks over different sizes of programs. To this end, we will want input programs of various sizes. Here, this is achieved through the concatenation of auto-generated functions. Functions differ wildly in size and structure, which was done purposely to simulate naturalistic programming.

4.2.2.2 Function Auto-Generation

As part of this testing mechanism, it was necessary to construct a script to auto-generate functions at random. The requirements were as follows:

1. Every function must have the same prototype; this makes constructing source files easier, as forward declarations and conditional calls to the functions are auto-generated.
2. Functions should vary in size and complexity (in this case, basic blocks).

Because there is randomness in how these functions are generated (and thus results may vary from system to system), the source for both the generating script and the auto-generated functions themselves has been included. Examples of functions generated are given in Appendix B.1.

4.2.2.3 Source File Generation

The purpose in generating these source files is to create programs of a given size, where size is measured in the number of functions. Each of these files will contain a similar vulnerability, but as the overall programs are different, different payloads will be required to exploit this vulnerability.

Each source file consists of the following:

1. A ‘header’, consisting of preprocessor directives and struct declarations.
2. Forward declarations for every function; functions may be shuffled to any arrangement to construct the source file, so they will need to be forward-declared.

3. Functions, including the `main` containing the vulnerability; `main` is itself auto-generated.

The `main` function differs slightly for every generated source for the following reason. All functions included in the source must be called from somewhere, or optimizations may remove them as dead code. Therefore, we include a call to each one in `main`. Incidentally, we do not wish to ever actually call these functions. For this reason, calls to them are made conditional on certain command-line inputs (inputs that are never given in testing). An example `main` is shown in Appendix B.2.

### 4.2.2.4 Procedure

Assume the following are available:

1. A snippet of vulnerable code.
2. `funcs`, A large corpus of auto-generated functions.
3. A routine to calculate the input payloads required to smash the stack of a vulnerable source.
4. A routine to check whether the stack of a given binary is smashed with given payloads.

We define the following parameters:

1. `max_funcs`, the maximum size of program to create, measured in number of functions inserted.
2. `versions_per_size`, the number of programs we are to create of every size.
3. `runs_per_technique`, the number of times we are to run each randomization technique on each base version.

The overall procedure is then as follows.

```plaintext
for i in 0..max_funcs do:
  for j in 1..versions_per_size do:
    construct source file as base version
      (put vulnerable main together with i functions chosen at random from funcs)
    compile base version
    calculate payloads to smash base version and spawn a new shell
    for t in randomization_techniques do:
      for k in 1..runs_per_technique do:
        re-compile base version with technique t
        check whether randomized version can be smashed using base version's payloads
```
4.2.2.5 Definition of Success

This is a note to describe the possible outcomes from attempting the attack on code that has been randomized. The following have been observed:

1. The attack succeeds, spawning a new shell.
2. The program exits normally, but without spawning a new shell: this is most commonly observed with DVI.
3. The program is terminated due to a segmentation fault (SIGSEGV): most commonly observed with FR and BBR.
4. The program is terminated due to an illegal instruction (SIGILL): most commonly observed with FR and BBR.
5. The program does not terminate, but instead falls into an infinite loop of unintended instructions: FR and BBR.

For the purposes of this evaluation, all but case 1 are considered to be a successful outcome. While it may be true that case 5 causes a problem of its own, the purpose here is to prevent the attacker from taking control of the host machine and executing arbitrary instructions.

4.3 Results

4.3.1 Parameters

The parameters used to obtain the results obtained in this section are as follows.

**Benchmarking**

- total_versions: 64
- runs_per_technique: 10
- readings_per_run: 4

**Effectiveness**

- max_funcs: 63
- versions_per_size: 20
- runs_per_technique: 100
4.3.2 Dummy Variable Insertion

DVI was tested with two different values for \texttt{max-allocs}:

- \texttt{DVI4}: \texttt{max-allocs}=3
- \texttt{DVI6}: \texttt{max-allocs}=5

In other words, using \texttt{DVI4} every function would have 0-3 variables inserted, while using \texttt{DVI6} every function would have 0-5 inserted.

4.3.2.1 Benchmarking

The results of benchmarking DVI against its unrandomized counterpart are shown in Table 4.1. Because different base programs are being compared, such figures only make sense when normalized. Therefore, the figures shown are normalized against those taken for the unrandomized version of each base version, for which the value is always 1.0.

DVI always leads to an increase in code size, but the increase is less than 1%. Unsurprisingly, it is greater for \texttt{DVI6} (0.40%) than for \texttt{DVI4} (0.24%). For \texttt{DVI4}, there is a one-in-four chance that no dummy variables will be inserted; for \texttt{DVI6}, this reduces to a one-in-six chance, so the code size is expected to be slightly greater. As it turns out, this increase is very slight.

The differences in retired instructions (increases of 0.19% for \texttt{DVI4} and 0.11% for \texttt{DVI6}) are so marginal as to be discarded as insignificant.

4.3.2.2 Effectiveness

Figures 4.1 and 4.2 illustrate the effectiveness of DVI at different program sizes. For both \texttt{DVI4} and \texttt{DVI6}, the proportion of successful attacks is plotted against the number of functions added. When average values are taken, the results appear completely inconsistent, without any clear trend being formed.

Scatter-plots of the same data reveal the reason behind this inconsistency (see Figures 4.3 and 4.4). At some sizes of program, there are some base versions generated where the attack cannot...
be blocked through DVI. If these were discounted, then the rate of successful attacks would average around 25% for DVI4 and 17% for DVI6, regardless of program size.

### 4.3.2.3 Assessment

Dummy variable insertion may generate a potentially limitless number of possible implementations of the same source. In practice, we bound the number so as not to get ridiculous stack frame arrangements. Even still, limitless possibilities reduce the likelihood of success of brute-force attack.
Chapter 4. Evaluation

One disadvantage is the potential for increase in both code size and stack frame size. A function’s local variables are allocated on the stack. A pointer is decremented such that enough space is left for each variable. If the function has other (non-dummy) local variables, then the pointer is simply decremented by a larger number, i.e. there is no increase in code size. However, if the function previously had no local variables, then instructions must be generated to insert the dummy variables.

Regardless of whether a function has non-dummy local variables, the insertion of dummies will inevitably cause an increase in stack-frame size. This is potentially problematic in the case of highly recursive programs. Even a small increase (say 1-2 32-bit variables) in stack frame size in a function that is very deeply recursive will increase the chances of overflowing the call
4.3.3 Function Re-Ordering

4.3.3.1 Benchmarking

Table 4.2 shows the results of benchmarking FR against its unrandomized counterpart.

FR features no measurable increase in code size. Of the three methods investigated, it has the largest performance hit. This is likely a result of a loss of locality benefits. Ideally, functions that call each other should be located in adjacent areas of generated code. Using a Knuth shuffle, functions could end up being absolutely anywhere in code, so (potentially) any such benefits are lost. However, at an average increase of 1.13% in retired instructions, this is still very low. The complexity of the task given to the program was very low; it is possible that, given a more complex task, a bigger disparity would be witnessed (see Chapter 5 for a note on further expanding these tests).

4.3.3.2 Effectiveness

Figures 4.5 and 4.6 illustrate the effectiveness of FR at different sizes. As before, the proportion of successful attacks is plotted against the number of functions added. When averages are taken, results show a sharp increase in effectiveness as program size increases. This is due to the factorial increase in possible orderings as each function is added. A program with \( n \) functions may be arranged \( n! \) possible ways by FR, but not all of these arrangements will successfully block an attack. The more possible re-orderings that are available, the more likely that some address crucial to the attack will be changed.

4.3.3.3 Assessment

An advantage of this approach is that there is no increase in code size. It scales very well; successful attacks in our prototyped system fall to around 1% on average with as few as 20 functions inserted. FR is potentially vulnerable to brute-force attack, but realistically, only for very small programs.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Size min</th>
<th>Size max</th>
<th>Size avg</th>
<th>Heap min</th>
<th>Heap max</th>
<th>Heap avg</th>
<th>Retired min</th>
<th>Retired max</th>
<th>Retired avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>1.0000</td>
<td>1.0001</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9796</td>
<td>1.0515</td>
<td>1.0113</td>
</tr>
</tbody>
</table>

Table 4.2: Benchmarking results for Function Re-Ordering.
The only notable disadvantage that emerges from testing is its performance penalty. However, it is very small.

### 4.3.4 Basic Block Re-Ordering

#### 4.3.4.1 Benchmarking

Table 4.3 shows the results of benchmarking BBR against its unrandomized counterpart.
TABLE 4.3: Benchmarking results for Basic Block Re-Ordering.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Size</th>
<th>Heap</th>
<th>Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td>min</td>
</tr>
<tr>
<td>BBR</td>
<td>0.9998</td>
<td>1.0047</td>
<td>1.0032</td>
</tr>
</tbody>
</table>

BBR will not always increase generated code size (in fact, it will occasionally reduce it), but features an average increase of 0.32% in retired instructions, which is comparable to DVI. It features almost no change in retired instructions from unrandomized versions.

### 4.3.4.2 Effectiveness

Examination of the effectiveness of BBR at different sizes (Figure 4.7) shows a reduction in successful attacks as the number of functions added increases. However, the reduction is slower than that observed with FR. With eight functions inserted, 50% of attacks are still successful; with 63 functions inserted, roughly 10% are.

![Average Effectiveness of Basic Block Re-Ordering](image)

**Figure 4.7:** Average effectiveness of Basic Block Re-Ordering

Figure 4.8 shows that, although the overall trend is good, there is a great deal of variation. Even as 60 functions are added, there are some programs for which no BBR arrangement will prevent an attack.

### 4.3.4.3 Assessment

BBR-randomized binaries feature barely any change in either code size or execution time from unrandomized counterparts. In terms of effectiveness, it scales reasonably well, but fares poorly for very small programs.
Figure 4.8: Effectiveness Basic Block Re-Ordering

4.3.5 Comparison

Figure 4.9 shows the average percentage difference from the unrandomized versions in retired instructions recorded for each technique. Figure 4.10 shows the relative effectiveness of the three methods.

FR is by far the best randomization method in terms of effectiveness. It does feature an average execution time performance penalty of 1.13%, and whether this is a problem will depend on the application; in a majority of cases, such a decrease in performance will go completely unnoticed.
There may be situations where BBR is preferable to FR. BBR is certainly ineffective for very small programs, but does improve with size. So for example, there may be a place for it in randomizing large-scale performance-sensitive code.

The effectiveness of DVI is consistent across program sizes. This means it does not scale particularly well. As program size increases, the code and stack frame size added by DVI will also increase, without providing any further increase in security.

To summarize: FR should be avoided in extremely performance-sensitive applications, BBR in very small programs, and DVI in very large or very recursive programs.

4.4 Chapter Summary

This chapter has discussed how my strategies for diversification were evaluated. It has discussed reasons for the testing methodology chosen, particularly in the context of existing work in the area. It has given the results of these tests, and interpreted them to give an assessment of each technique. The next chapter will discuss the potential for further work in this area.
Chapter 5

Further Work

5.1 Amendments to Existing Randomization Techniques

Testing revealed a number of features that might be addressed by changes to the randomization techniques that have been implemented.

In Dummy Variable Insertion, it would be useful to do an analysis of functions that are potentially recursive. Ideally, this should include those that are mutually recursive. The idea here is that this particular technique inevitably leads to an increase in stack frame size. For very deeply recursive programs, this might be highly undesirable behaviour. It results in higher memory usage, up to and including a potential for stack overflow. Further work would attempt to determine to what extent this may be a problem, and how best to address it.

In Function Re-Ordering, it would be useful to explore the idea of restricted shuffling. As noted, the use of a Knuth shuffle means that functions may end up anywhere in generated code, meaning any possible benefits of code locality are potentially lost. It might be beneficial to adapt a more restrictive algorithm, such that functions could not be shuffled too far out of place. This pre-supposes that the functions are already placed in code so as to best benefit from locality (most likely as a result of an earlier compiler optimization).

5.2 New Randomization Techniques

This project implemented only three techniques, out of a large number of possible ways to randomize a program at compile-time. In particular, it would be useful to compare these techniques to those at different levels of granularity. Only module-based and function-based ones were implemented here.
At a finer level of granularity than those found here would be those at basic block level. This would involve analysing and randomizing each basic block, in other words, randomizing something about individual instructions. Such techniques could include:

- Instruction selection.
- Instruction scheduling.
- Garbage code insertion.

At the opposite end in terms of granularity, whole-program techniques were considered out-of-scope when limiting myself strictly to compile-time strategies, but they could be explored as part of a broader analysis.

### 5.3 Further Testing and Analysis Techniques

#### 5.3.1 Benchmarking

Since benchmarking has been covered extensively in related work, my decision was to focus on effectiveness evaluation instead. As a result, there are a few areas of this testing mechanism that would benefit from further development.

Firstly, no test for stack memory usage was ever constructed. This was not done because of time constraints, but would be very useful for comparing behaviour.

Secondly, the benchmark tests themselves are small and straightforward. It would be desirable to have a more complex test, i.e. to have sqlite3 perform something above the most trivial tasks. This might give more noteworthy results.

#### 5.3.2 Effectiveness

As noted in Evaluation (see Chapter 4), there was an oddity concerning the effectiveness of Dummy Variable Insertion, specifically that there appear to be pathological cases where it is unable to prevent attacks, regardless of how many variables are inserted and where. Further analysis would examine the relevant source files and determine the root cause.
5.4 Chapter Summary

This chapter has discussed the potential for further work in this area. It outlines the possible exploration of amendments to existing randomization techniques, new randomization techniques, and enhanced testing and evaluation methods. The next chapter will summarize my project and discuss the conclusions reached.
Chapter 6

Conclusions

Investigation into the background of this project has shown that the problem of software monoculture is ongoing. Examples given in this report have shown what types of attacks are possible, and how a lack of diversity in distributed binaries is contributing to the problem of security worldwide.

To investigate how this problem may be mitigated, techniques for compile-time application binary randomization have been explored. This sought to test the theory that the negative effects of a monoculture may be diminished with binary diversity, in that different versions will not all be susceptible to the same attack. Implementations have been provided for three techniques, Dummy Variable Insertion, Function Re-Ordering, and Basic Block Re-Ordering.

Proof of concept has been implemented in a real-world production-grade compiler, LLVM. This shows that this is a practicable rather than purely theoretical solution.

The three solutions have been scripted to hook into LLVM, so that properties may be tested about the quality of the binaries the augmented tool-chain produces. These tests showed that, for the metrics benchmarked and using a simple test, there were no particularly large penalties for using randomization. A mechanism for experimentally testing the effectiveness of each technique in blocking a specific type of attack has been constructed. It shows good results, specifically, that two of the methods improve with program size. My method contrasts with the approach taken by related approaches to assessing the effectiveness of binary randomization, which are almost exclusively analytical.

In summary, all project objectives (theory, design, implementation, and testing) are deemed to have been met.
Appendix A

Sample PLT Attack

Based on Stack Smashing on a Modern Linux System [jip12].

We note the source file eggshell.c. This program features two functions. One of these contains a buffer overrun; another makes a system call that spawns a shell. It takes two user-provided files as input. The files are read into two buffers, and those buffers are then copied into two `item` structs.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define SIZE_MAX 48 // Maximum size of item's data buffer
#define SIZE_BUFFER 56 // Maximum size of the string buffer to be entered

struct item{
    char data[SIZE_MAX];
    struct item * next;
};

void spawn_shell(){
    exec1("/bin/bash", "/bin/bash", "-p", NULL);
}

int main(int argc, char ** argv){
    char * filename1 = argv[1];
    char * filename2 = argv[2];

    FILE * file;
    char buffer1[SIZE_BUFFER], buffer2[SIZE_BUFFER];

    file = fopen(filename1, "r");
    if(file){
        fread(buffer1, 1, SIZE_BUFFER, file);
        fclose(file);
    }
```
file = fopen(filename2, "r");
if(file){
    fread(buffer2, 1, SIZE_BUFFER, file);
    fclose(file);
}

struct item item1, item2;
item1.next = &item2;
item2.next = &item1;

memcpy(item1.data, buffer1, SIZE_BUFFER); // Potential for overflow!!
memcpy(item1.next, buffer2, SIZE_MAX); // This is the correct size

_IO_putc('*', stdout);
}

From the source, we note that a buffer overrun occurs in main (at the first memcpy). Once we go beyond the real bounds of item1.data, we will be writing into item1.next. The second memcpy writes to the location pointed at by item1.next, which should be item2, but may have been altered by the overrun. Therefore, the attacker has the opportunity to write SIZE_BUFFER number of bytes to anywhere they like in memory.

Later in the code, we note a call to the library function _IO_putc().

The basic idea of this attack is as follows. On the first memcpy, we will overwrite item1.next with the address of the GOT.PLT jump slot of _IO_putc(). On the second memcpy, we will overwrite that jump slot with the address of spawn_shell().

At run-time, prior to those calls to memcpy the state of the system is approximately what is shown in Figure A.1. If we, as the attacker, construct our attack correctly, the state will be Figure A.2 following the memcpy calls.

Now, when we come to the call to _IO_putc(), its jump slot is looked up in the GOT.PLT. It retrieves what it believes is the address of _IO_putc(), but is in fact the address of spawn_shell(). Executing the call to spawn_shell() creates a shell with the permissions of whatever user ran the program. So for example, if this user had root permissions, the attacker will have a root shell.

To construct the required payloads, we start with a binary compiled under conventional compilation from this source file. We disassemble the binary and find the addresses we need: _IO_putc@got.plt (the jump slot address of _IO_putc()) and spawn_shell(), together with the location of exec1. exec1 is another library function, the one that makes a call to start /bin/bash. These can be found with simple, common tools like objdump and gdb.

Such a convenient function would not always be available in a typical application binary. It is possible [sic11] to figure out where /bin/bash actually is at run-time. This function is included in the example to make the process more clear.
Appendix A. Sample PLT Attack

The first payload consists of 48 bytes of padding followed by the jump slot address. The second payload consists of two parts. The first is the address of spawn_shell(). The second is the location of exec1; when we write into the GOT.PLT on the second memcpy, we must ensure that this location is preserved.

The above attack has been implemented successfully as part of the research for this project. Two separate systems were targeted:

Appendix B

Auto-Generated Code for Effectiveness Testing

B.1 Examples of Auto-Generated Functions

```c
/* function auto-generated using seed: 2331101 */
void f79(int x, int y){
    int z = 1;
    x = z * z;
    if(x == y){
        z = y * z;
    }
    printf("%d", z);
}

/* function auto-generated using seed: 1640816 */
void f319(int x, int y){
    int z = 1;
    y = z - x;
    if(z == x){
        y = y + x;
        if(y <= x){
            y = x - y;
            if(z != z){
                z = z + y;
                y = y - x;
            }
        }
    } else{
```
Appendix B. Auto-Generated Code for Effectiveness Testing

z = y * y;

if (z > x)
    z = z - x;
y = y + z;
z = z - x;

z = y + x;

y = x - x;

printf("%d", z);

B.2 Sample Main

int main(int argc, char ** argv)
{
    char * filename1 = argv[1];
    char * filename2 = argv[2];

    if (argc == 5)
    {
        int x = atoi(argv[3]);
        int y = atoi(argv[4]);

        f242(x, y);
f60(x, y);
    }

    FILE * file;
    char buffer1[SIZE_BUFFER], buffer2[SIZE_BUFFER];

    file = fopen(filename1, "r");
    if (file)
    {
        fread(buffer1, 1, SIZE_BUFFER, file);
fclose(file);
    }

    file = fopen(filename2, "r");
    if (file)
    {
        fread(buffer2, 1, SIZE_BUFFER, file);
fclose(file);
    }

    struct item item1, item2;
    item1.next = &item2;
    item2.next = &item1;

    memcpy(item1.data, buffer1, SIZE_BUFFER); // Potential for overflow!!
```c
memcpy(item1.next, buffer2, SIZE_MAX); // This is the correct size
_IO_putchar('+', stdout);
```
Bibliography


