A Systematic Approach to Automated Software Diversity Using Unison

PATRIK KARLSTRÖM
A Systematic Approach to Automated Software Diversity using Unison

Patrik Karlström

Master in Embedded Software
Date: June 21, 2018
Supervisor: Benoit Baudry
Examiner: Christian Schulte
Swedish title: Ett Systematiskt tillvägagångssätt för Automatiserad Mjukvarumångfald med Unison
School of Electrical Engineering and Computer Science
Abstract

Unison is a tool that combines instruction scheduling and register allocation as a single combinatorial problem and solves it using constraint programming, which is a programming paradigm for systematically solving combinatorial problems.

Automated software diversity is the process of automatically providing diverse executables in an effort to break so called gadgets, which are short instruction sequences that together make up an attack vector. Attacks that utilize gadgets rely heavily on the arrangement of the code in the executable. By providing a population of executables with equivalent functionality but different arrangements an adversary must construct a unique payload for each executable. The idea is to mount a proactive defense against adversaries and limit the reusability of each constructed payload.

The results when using Unison to systematically generate diverse executables show that the number of possible pairwise distinct executables is often larger than 1000000, even for small functions (less than ten instructions). Using Unison to force the executables to differ in a particular way is simple to implement, only a handful lines of code. One strategy evaluated in the experiment resulted in that the most frequent gadget only appeared in 24% of versions, and 82% of the gadgets only appeared in one program version each. However, future work is required before anything consumer oriented can be evaluated, in part because Unison does not support the x86 architecture.
Sammanfattning

Unison är ett verktyg som kombinerar instruktionsschemaläggning och registerallokering som ett enat kombinatoriskt problem och löser det med villkorsprogrammering, vilket är en programmeringsparadigm för att systematiskt lösa kombinatoriska problem.


Att använda Unison för att systematisk generera varierade exekverbara filer visar att antalet möjliga, parvis distinkta exekverbara filer är ofta större än 1000000, även för små funktioner (mindre än 10 instruktioner). Att använda Unison för att tvinga de exekverbara filerna att skilja sig på specifika sätt är enkelt att implementera, bara en handfull rader kod. En utvärderad strategi i experimentet resulterade i att den gadget med högst frekvens återfanns i 24% av alla programversioner, och 82% av alla gadgets återfanns i endast en programversion var. Dock så krävs mer arbete innan något konsumentorienterat kan utvärderas, delvis för att Unison inte stödjer processorarkitekturen x86.
Acknowledgements

I am very grateful to both Benoit Baudry and Christian Schulte at the SCS department of KTH for their support, without which this project would not have been possible.

A special thanks is owed to Roberto Castañeda Lozano, researcher at the Computer Systems Laboratory of SICS and the main author of Unison, who has provided invaluable assistance with both devising and implementing my model. Thank you, Roberto!
# Contents

1 Introduction .......................... 1
  1.1 Problem Statement ..................... 2
  1.2 Motivation ............................ 3
  1.3 Methodology ........................... 4
  1.4 Ethics and Sustainability ............... 4
  1.5 Overview ............................. 5

2 Background .......................... 6
  2.1 Software Diversity ..................... 6
  2.2 Return Oriented Programming .............. 7
    2.2.1 Example of a code re-use attack .......... 8
  2.3 Constraint Programming ................ 13
  2.4 Unison ............................... 16

3 Diversification with Unison .......... 18
  3.1 Automatic Diversity Synthesis with Unison .... 18
  3.2 Cost ................................ 19
  3.3 disUnison ............................ 19
    3.3.1 Enumeration ....................... 20
    3.3.2 Registers ........................ 20
    3.3.3 Instruction Schedule ............... 21
  3.4 Branching Strategy .................... 22
  3.5 Sampling Rate ........................ 22
  3.6 Target Architecture .................. 23

4 Experimental Setup ................. 25
  4.1 Data Set ............................ 25
  4.2 Metrics ............................. 26
  4.3 Generation Process ................... 26
## CONTENTS

5 Results and Discussion
  5.1 Generation Time ............................................. 29
  5.2 Cost ............................................................. 30
  5.3 Surviving Gadgets ............................................. 32

6 Discussion
  6.1 Discussion ...................................................... 35
  6.2 Threats to Validity ............................................ 36
  6.3 Future Work ....................................................... 37
  6.4 Conclusion ....................................................... 38

Bibliography .......................................................... 39

A Functions
  A.1 Functions and Statistics ................................. 43

B ROP Exploit .......................................................... 45
Chapter 1

Introduction

The field of software diversity is concerned with researching the causes and effects of diversity in software and software engineering, both in the result and the process. As an example consider web browsers. There are many different implementations of what is essentially the same functionality. In the context of security vulnerabilities of one browser does not necessarily carry over to another [1].

One of the applications of software diversity is as a defense to code re-use attacks [1]. Code re-use attacks refers to attacks that diverts program control flow to re-use already present code in an unintended manner [23]. For example if an adversary gains access to a process’ stack the address to jump to after a return instruction can be overwritten and the adversary can choose which instruction to jump to. Shacham [24] extended this concept and introduced what is called return-oriented programming. Return oriented programming is a code re-use technique were the program control flow is diverted to a chain of short instruction sequences, dubbed a gadget. By carefully choosing these gadgets an adversary can achieve arbitrary code execution.

However, the adversary relies on the fact that the chosen instruction sequences are always equivalent across all binary files. In other words an equivalent sequence of instructions is located at the exact same offset. If, for example, everyone would run their own, structurally different but functionally equivalent version of the Firefox web browser an adversary would have to disassemble all variations and construct a unique payload for every target. Thus, techniques that provide diversity between executables and/or runtimes have been researched as a defense to these code re-use attacks [1].
To understand how two executables can be structurally different but functionally equal consider an example pertaining the instruction schedule. Say there are two independent instructions, one of which is part of a gadget. By swapping the place of these two instructions the gadget is now different, but since the instructions are independent program functionality is the same. There are many decisions taken by the compiler that are not necessarily the only valid decision, and if one could explore all combinations of decisions one could systematically generate the entire population of diversely arranged but functionally equivalent executables.

Constraint programming is a paradigm for solving combinatorial search problems wherein the relationship between variables are expressed as constraints. It solves problems such as scheduling, vehicle routing and, in our case, compiling [21, 13]. For example, consider the problem of scheduling working shifts in a store. The variables could be the start and end time of a shift for one person, and the constraint could be that there are always two employees present in the store and no one works for more than 8 consecutive hours.

A constraint solver takes a problem such as the example above and finds a value for all the variables such that all constraints are satisfied, or in the case when no mapping that satisfies all constraints exists proves that such is the case [21]. The process of finding these values consists of searching the combinatorial space and either implicitly or explicitly discarding combinations that cannot be solutions, i.e. does not satisfy the constraints.

Unison is a tool that uses a constraint solver to schedule instructions and allocate registers as a single, integrated problem. It operates as a part of the LLVM backend, and is thus not a standalone compiler. Unison is potentially optimal in the sense that given enough time, it can generate the optimal instruction schedule and register allocation. It finds this solution by searching the combinatorial space in a continuous pursuit for a solution that is better than the last [13].

1.1 Problem Statement

It might feel intuitive to provide software diversity by somehow randomizing a part of an executable, and in fact many have [1, 9, 25], but this approach comes with a three key challenges: It is difficult to make
guarantees about the resulting population, proving the correctness of the code transformations is challenging and there is no fine-grained control of the process.

When generating the executables with a random element in the process you cannot make any guarantees regarding the resulting population. If you generate two executable you cannot with certainty tell that they are in fact sufficiently different without verifying it. The risk of them being equal is most likely very small for just two versions, but what if 10000 different version are generated? Or 1000000? At some point the pigeonhole principle comes into effect, in which case case it would be better to re-use the already generated code.

By instead systematically generating different variations those sort of guarantees can be made. If two equal executables are never generated it follows that they are all pairwise distinct. It also follows that if the code generator is run to termination, all pairwise distinct versions are generated. There is also opportunity to define exactly what pairwise distinct entails. The nature of constraint solving lends itself well to this application. This is where Unison comes in. In addition to its main purpose Unison has potential for software diversity purposes.

This thesis aims to be an early exploration of a systematic approach to automated software diversity using a constraint solver, specifically the Unison tool. It aims to evaluate both the code generator and the generated code. This leads us to the hypothesis that

“Systematically generating diverse executables using a constraint solver is at least as good as basing the diversification process on randomization.”

1.2 Motivation

As mentioned, it is difficult to prove the correctness of code transformations. Perhaps the most compelling reason to use a constraint solver for software diversity is that for a combination to be considered a solution, all constraints must be satisfied. Regardless of what unorthodox approach we want to try out we do not need to consider it’s effect on program functionality, since the constraints that ensure a valid program is generated must still be satisfied. This property makes implementing new techniques relatively simple. Modifying code while retaining functionality is no easy task, and this simplifies it greatly.
With the systematic approach there is full control of what changes between versions, and how much it changes between versions. Instead of introducing randomness into some part of the compilation process and more or less hope it generates diverse code with low overhead the systematic approach offers the possibility of properly reasoning about the process and steering it specific manners. It also yields a greater control of the quality of the generated code. Limits can be dynamically set in place so that it does not exceed e.g a certain number of instructions.

1.3 Methodology

This project is an early exploration of the potential benefits of using a constraint solver for automated software diversity. Specifically, the research will be carried out by modifying the Unison tool to support generating multiple outputs. Exactly how these outputs should differ to generate the most diverse population is outside the scope of this project. With that said, for the purposes of this thesis the modification of Unison will include three different options of how the resulting code should differ. Both the modified code generator as well as the resulting population of each of the three options will be evaluated. This lends itself well to an experimental approach.

The purpose of the experiment is to evaluate both the code generator and the resulting code. The evaluation of the code generator will be performed on both quantifiable and non-quantifiable metrics (such as ease of implementation). The resulting code, which will be a population of functionally equivalent but differently arranged code, will be evaluated in a purely quantifiable way.

1.4 Ethics and Sustainability

Developing defenses in the field of software security does not generally offer many ethical controversies. It is in principle always good to offer defense against ominous deeds. However, generating multiple executables on each invocation of a compiler would lead to sustainability impacts.

Jackson et al. [10] describes the architecture for a solution like this when deployed on a large scale. It involves an "app store" (e.g a pack-
age manager repository or a smartphone app store) where a user asks for a binary and each user is given a unique variant. The two options are to either generate a new version when one is asked for or to generate as many versions as feasible beforehand, storing them somewhere and simply providing one at user requests. Both of these approaches involves more computation than just compiling a single binary and providing it to all users. In the latter case a lot of storage is also involved.

Initially either of these factor might not seems like much but requiring additional computation and potentially storage that scales linearly with users can become a problem for popular applications, such as certain web browsers or common software libraries. The extra computation and storage requires power which is costly not only in monetary terms but also environmentally. Unfortunately, this is an unavoidable consequence of a large scale deployment of automated software diversity.

1.5 Overview

This thesis is divided into five chapters. The first chapter provides an overview and introduction of the project background and goal as well as the research method and ethics and sustainability issues.

Chapter 2, background, presents software diversity and why it is desirable. Also presented is how the systematic approach will be taken in the form of Unison and constraint programming.

Chapter 3 and 4 of the thesis consists of, respectively, presents and explains the implemented model and the experimental setup. Chapter 5 presents the results of the experiment. The last chapter discusses the shortcomings of the experiment, the future work required and the conclusion.
Chapter 2

Background

This chapter will introduce the necessary background to this project. It will begin by introducing what software diversity is and why it is desirable. There will also be a high-level description of Unison, the primary tool used throughout this project. As pre-requisite for Unison there will also be a section about constraint programming.

2.1 Software Diversity

Software diversity is the study of diversity in software and software engineering. It is a diverse discipline and there is a wide range of research all with different focus and set of goals. The one common factor is that they all explore the potential benefits of diverse engineering [1].

Software diversity has a myriad of applications ranging from fault-tolerance to re-usability to security [1]. For security purposes it is a relatively common approach to certain issues such as memory error exploits and different approaches have been taken. For example Chew and Song [5] randomizes the interface between user space applications and the operating system, Bhatkar, Sekar, and DuVarney [3] introduces a level of indirection to function call to randomize static data at the C source code level and Bhatkar, DuVarney, and Sekar [2] obfuscates the code and data of an executable by randomizing the absolute locations.

Failing to defend against certain memory errors can introduce new attack vectors for a determined adversary. Some of these attacks are based on the availability to reverse engineer a vulnerable executable, where software diversity can once again prove an effective defense.
To defend against reverse-engineering based attacks Wartell et al. [25] dynamically determine the addresses of basic blocks at load time, Pappas, Polychronakis, and Keromytis [20] performs in-place code randomization to make each executable look different without incurring significant overhead and Homescu et al. [9] implements a library to transparently diversify code compiled and executed by JIT-compilers.

Many more strategies and techniques exist for what is known as automated software diversity and Larsen et al. [11] provides an overview of different goals, attacks, defenses and deployment methods. Automated refers to the fact that a human is not involved in the process (other than devising the process).

What all of these techniques have in common is that they introduce some kind of chance into the development toolchain, which as mentioned will not be the approach taken in this paper. Instead the focus will be on systematic and deterministic generation of static diversity. Static diversity entails generating several, diverse executables; In contrast to dynamic diversity which involves providing diverse run-times [1].

The motivation is that by employing diversification techniques one can break the reactive nature of software security and shift the advantage from the attacker to the defender [10].

### 2.2 Return Oriented Programming

Shacham [24] introduces return-oriented programming (ROP) as an extension of the return-to-libc [7] attack that allows arbitrary code execution without the limitation of calling complete functions. All examples in this subsection is only concerned with the x86 and x86-64 architecture as this is the architecture were the concept was first introduced and where it is most applicable.

A typical return-into-libc attack involves somehow diverting the program control flow to transfer control to code that already exists in the program’s address space, it could e.g be the system() library function, and use it in an unintended manner. For example an adversary could utilize a buffer overflow to overwrite the return address on the stack so that it instead points to a function call that leaks sensitive information. By first jumping to a sequence of instructions such as "pop reg; pop reg; ret;" it is possible to chain together several of these func-
tion calls, performing more complex operations [19, 18].

Shacham [24] builds upon this concept by instead diverting the control flow to just these short sequences of instructions, usually just two or three instructions, typically ending with a "ret" instruction. These sequences are dubbed gadgets and by carefully choosing and chaining them together one can achieve arbitrary code execution [24].

For instruction set architectures with varying instruction sizes, such as x86 and x86-64, there can be gadgets not explicitly placed there by the compiler. Shacham [24] explains them with an analogy to the English language: the word "dress" can be found in "Address" and the word "head" can be found in "The Address" depending on where you start and stop reading. When instructions can be of varying lengths then the same sequence of bytes can be interpreted as different instructions depending on where one starts reading. If you can find a hidden "ret" instruction you can potentially find a whole gadget not explicitly placed there by the compiler. The x86-64 architecture has 4 "ret" instructions and the most basic one is only a single byte [26], making hidden gadgets frequent.

The reason that the "ret" instruction is convenient is because it transfers control flow to the instruction pointed to by the value at the top of the stack. That is, the "ret" instruction reads the top of the stack as a memory address and transfers the control flow to continue from that address. If an adversary can control the stack he or she can also control the control flow. Gadgets can end with other instructions depending on what is desirable.

2.2.1 Example of a code re-use attack

As an example attack consider the x86-64 version of the "split" challenge on ROP emporium\(^1\). The challenge consists of two files, an executable and a "flag.txt" file. The goal is to provide a payload to the stdin of the binary that executes the command "/bin/cat flag.txt". It is a simple challenge in the sense that the string "/bin/cat flag.txt" and a call to the `system()` function (which executes a shell command) are already present in the binary. The intended behaviour of the binary is to execute `system()` with the argument "/bin/ls".

\(^1\)https://ropemporium.com/challenge/split.html (visited on 20/06/2018)
Approach

In order to construct and mount or attack we have a few boxes to check. Namely we must:

1. Identify a vulnerability.
2. Disassemble the program, find the available gadgets, function calls and strings.
3. Construct our payload

As previously mentioned, in this case we have crucial knowledge about the executable beforehand. We know that a call to the `system()` function and the string "/bin/cat flag.txt" are already present in the binary. However, we don’t know where.

Identifying a vulnerability

When executing the executable we are met with a prompt for user input. If we enter a long string, e.g 100 "A" characters, the program raises a segmentation fault. This tells us that there is a buffer overflow somewhere that overwrites the return address on the stack.

We can exploit this buffer overflow vulnerability in order to call the `system()` function in an unintended manner. By disassembling the executable and analyzing the code we find that in this case the vulnerability is that a buffer of size 32 bytes is allocated and 92 bytes is read into it. The string will grow from lower memory addresses to higher while the stack grows from higher addresses to lower. In other words, by writing 92 bytes to the buffer, 60 bytes of data allocated lower in the stack is overwritten. Included in these 60 bytes is (in this case) the return address and by overwriting this to the instruction that invokes the `system()` function call we can call the function outside of the normal program control flow.

The calling convention of the AMD64 Linux Kernel is to load the arguments to the kernel interface in registers rdi, rsi, rdx, r10, r8 and r9 in that order [16]. In other words, we want the rdi register to point to the "/bin/cat flag.txt" string before we call the `system()` function. In order to do this we must find the sequence "pop rdi; ret" somewhere in the binary. If we manage to execute this gadget when the address of the "/bin/cat flag.txt" is at the top of the stack and the address of the
call to system() is at the penultimate position we will execute the shell command and print the contents of "flag.txt". The "pop rdi" instruction will load the value at the top of the stack, which is the address of the "/bin/cat flag.txt" string, into the rdi register and increment the stack pointer (remember that the stack grows downwards). The "ret" instruction will then read the value currently at the top of the stack, which is the address to the invocation of the system() function, and transfer the program control there. The system() call will read the argument in the rdi register and execute the command.

Putting this together, we want to overwrite the return address of the vulnerable function with the address of our gadget (i.e. the address of the "pop rdi" instruction). When control is transferred to the gadget the top of the stack should be the address of the "/bin/cat flag.txt" string, followed by the address of the system() call.

Without the previous knowledge of the system call and the argument string we would have had to find these pieces by disassembling the program.

Disassembly and gathering all pieces of the puzzle

We know what we want to do and we have identified an attack vector. However, a few pieces of the puzzle still remain. We must find the offset of the return address on the stack, i.e. how must the payload be constructed in order to overwrite the return address with the value we want. We must also find exactly what values should be placed on the stack.

To find the offset of the return address we can run the program and enter garbage to the stdin, e.g "AAAAABBBBBCCCCDDDEEEFFFF" and so on. If we overwrite the return address the program will most likely raise a segmentation fault (as it did when we identified the buffer overflow), and from the core dump we can read what address it tried to divert control flow to. If it is e.g "0x4545454546464646" (which is the ascii encoding of "EEEEFFFF") we know that by padding our payload with 16 characters we overwrite the return address. There are more clever ways to do this but the simple way also works. In this case the padding required is 40 bytes.

The address of the gadget (the "pop rdi" instruction) to be used is 0x400883, the address of the "/bin/cat flag.txt" string is 0x601060 and the address of the system call is 0x400810. All of these addresses as
well as the required padding were found using the radare2\textsuperscript{2} reverse engineering framework.

**Formulate and mount the payload**

We want to place the address of the gadget at the return address, followed by the address of the "/bin/cat flag.txt" followed by the address of the system call. We know from forcing a segmentation fault that to overwrite the return address we need 40 bytes of padding. We also know that since the input buffer and the stack grows in different directions the stack will read our values in the order we write them.

The hexadecimal encoded payload for this example is thus:

\begin{verbatim}
4141414141414141414141414141414141414141414141414141414141
4141414141414141414141830840000000000060106000000000001008400
00000000
\end{verbatim}

which includes 40 "A"s (0x41) as padding, followed by the little endian representation of the gadget address, the string address and the address of the system function call. Each of which is 8 bytes (16 hexadecimal digits). When this payload is posted the stack will look as depicted in Figure 2.1.

The function requesting the input will reach its "ret" instruction, 0x400883 will be popped from the stack and the control flow will be transferred there. The stack will now look like presented in Figure 2.2.

The "pop rdi" instruction will be executed and 0x601060 will be loaded into register rdi and the stack pointer is incremented. 0x400810 is now at the top of the stack and when the following ret instruction is executed the control flow will be transferred to the instruction at address 0x400810, which is a call to the system function (see Figure 2.3).

The system function, whose argument is located in the rdi register, will read the "/bin/cat flag.txt" string and execute the command and the contents of flag.txt will be printed.

Appendix B contains the source code of a python script that executes this attack.

This "attack" does not actually do anything malicious, as it is just an example. As mentioned it was also fairly simple in the sense that the argument to be passed to `system()` was already present in the binary.

\footnote{\url{https://github.com/radare/radare2} (visited on 20/06/2018)}
One could supply this argument with the payload. In fact, including a "/bin/sh" string in the payload and providing it as an argument to the `execve()` function is one of the examples described by Shacham [24] when introducing the concept of return-oriented programming. In this example Shacham [24] performs a number of computations to set up the arguments (including calculating the address of the "/bin/sh" string on the stack), showing that if one is clever enough there are few limits to what code can be executed simply by re-using the code already present in the binary.

**Defending against code re-use attacks**

Defending against a technique such as the one exemplified above is difficult. Gadgets will always be present in every binary. But as mentioned software diversity can provide some manner of protection. The idea is that since an adversary is reliant of the exact addresses of these
gadgets a way to limit the possible attack surface is by creating a population of binaries who share as few of these gadgets as possible, and then distribute these evenly among all users. An adversary would then have to identify which version the target is running, obtain and analyze that version and create a tailored payload. This payload can then not be re-used on any other version of this program.

For example, the payload crafted for the example above only works on that binary. If a compiler could generate two binaries, the one "attacked" above and one where the system() call is shifted to the address 0x00400808 a new, different payload would have to created for the latter binary. The payload crafted above would not work if the system() call is not located at 0x00400810. In fact, the payload crafted in the example would not work if the address of the gadget, the system call or the address of the command string changed.

### 2.3 Constraint Programming

Constraint programming is a programming paradigm for solving combinatorial problems. By declaring all variable’s possible values and their constraints, the relationship between them when part of a solution, a constraint solver can search the combinatorial space and find
Figure 2.3: The contents of the stack, the stack pointer, pointer references and contents of the rdi register after the "pop rdi" instruction.

an assignment of variables to values that is consistent with the constraints. A constraint solver effectively explores different possible combinations systematically, by a potentially incomplete local inference (also known as constraint propagation) or more commonly a combination of the two [21].

The default constraint solver of Unison (which is the primary tool used in this paper. See 2.4) is Gecode\(^3\) [13], which is a constraint solver that interleaves systematic search algorithms with constraint propagation [22].

When propagating constraints the Gecode solver searches all variable’s domains and removes variables that are in conflict with the constraints [22, Section 23.1]. For example, given two variables (and their corresponding domains) \(x \in \{0, 1, 2\}\) and \(y \in \{0, 1, 2\}\) and the constraint \(x > y\) constraint propagation can determine that \(x \in \{1, 2\}\) and \(y \in \{0, 1\}\) are the only combinations consistent with the constraint.

When constraint propagation is finished there are three possible states:

---

\(^3\)www.gecode.org (visited on 20/06/2018)
1. One or more domains could be empty, proving that no solution exists.

2. All domains could be of size 1, indicating that there exists only one possible value for every variable, and thus we have found a solution.

3. One or more variables have multiple values in their domain.

For situation (1) and (2), either a solution is found or we have proven that a solution does not exist (within the local search space). The possible solutions in the above example are \( x = 1, y = 0 \) and \( x = 2, y \in 0, 1 \).

In the latter situation (3) the Gecode constraint solver splits a variable’s domain into two or more subsets, creating a search tree where each branch represents reducing the variable’s domain to a particular subset [22, Section 8]. For the above example branching could entail trying one case where \( x = 1 \) and one case where \( x \neq 1 \). By committing to a branch the constraint solver can once again perform constraint propagation and repeat the process. However, if branching has taken place and the solver reaches situation (1) it can go back up the tree and explore a different branch. If situation (2) is reached it can still backtrack but with the added option of adding more constraints based on the newly found solution. For example, if the \( x = 1 \) branch was taken the \( x = 1, y = 0 \) solution is found, in which case we could post the constraint \( x > 1 \). Constraining based on previously found solutions in the search tree is done with the branch and bound search engine [22, Section 9] in Gecode.

Branch and bound is an efficient strategy to find the optimal solution to a combinatorial problem which in essence constitutes comparing potential solutions to the currently best found solution, keeping the better of the two [6].

We adopt a similar strategy, but instead of constraining solutions to be better we require them to be different. In addition, no solutions are discarded. In the context of Unison a solution refers to a valid instruction schedule and a valid register allocation.
2.4 Unison

Unison is an open-source\textsuperscript{4}, potentially optimal tool that performs integrated register allocation and instruction scheduling using constraint programming. It can be used as an alternative or complement to the algorithms currently in place by compilers such as GCC\textsuperscript{5} and LLVM\textsuperscript{6}. In particular there already exists a driver for LLVM which accept input in the form of LLVM MIR \textsuperscript{13}.

LLVM MIR is a human readable, YAML\textsuperscript{7} serialized format of the LLVM machine specific representation \textsuperscript{17}. The machine specific representation is used by the LLVM code generator to emit assembly for a specific architecture \textsuperscript{4}.

Unison models the problems of register allocation and instruction scheduling as a single constraint programming problem and solves them simultaneously using a branch and bound algorithm \textsuperscript{13, 15, 14}.

To allow for further optimizations than the current LLVM code generator does Unison also introduces optional copies and alternate temporaries \textsuperscript{15}. These two techniques increases the number of possible executables that can be generated by introducing more combinations into the constraint satisfaction problem. While primarily used for code optimization they can also be used for diversity.

The implementation strategy for these techniques is to create an abstract operation, which can potentially be implemented by several instructions, or not at all. In addition, the operands are considered constraint variables that can be connected to one out of a set of temporaries \textsuperscript{13}, or not be connected at all. By creating a level of indirection between operations and instructions as well as their corresponding operands and temporaries the problem can more easily be modeled as a constraint programming problem where operations and operand are the variables with instructions and temporaries as domains, respectively.

With Unison we have control over every decision made related to instruction scheduling and register allocation, and more importantly we can base decisions on previously found solutions. Unison works on the function level and a solution in this case refers to an instruc-

\textsuperscript{4}https://github.com/unison-code/unison (visited on 20/06/2018) 
\textsuperscript{5}https://gcc.gnu.org/ (visited on 21/06/2018) 
\textsuperscript{6}http://llvm.org/ (visited on 21/06/2018) 
\textsuperscript{7}http://yaml.org/ (visited on 21/06/2018)
tion schedule and register allocation for the given function that is executable on the target architecture.
Chapter 3

Diversification with Unison

In this chapter the relevant parts of the Unison model, the modification to the models that will be implemented and the experimental protocol will be presented.

The first section, Automatic Diversity Synthesis with Unison briefly presents the Unison model and how to implement our extension of it. The disUnison section describes how we extend the model to provide software diversity, and the Experimental Protocol section describes how the experiment will be conducted.

3.1 Automatic Diversity Synthesis with Unison

In its current state Unison accepts input in the form of the LLVM MIR (see Section 2.4) of a single function [13] and outputs LLVM MIR of the same function with allocated registers and scheduled instructions. It is then the job of llc, the LLVM static compiler to emit architecture specific assembly, which can be passed through a native assembler and linker to generate an executable [12].

As described in Section 2.4 the problem of integrated register allocation and instruction scheduling in Unison is modeled around operations and operands. The problem consists of around 20 variables (see 2.3) in total so only the ones relevant for the experiment will be presented. These variables describe how the operations and their operands relate to the actual instructions, temporaries, registers and issue cycles.

The only constraint solver currently supported by Unison is Gecode
A branch and bound search can be implemented in Gecode by using a branch and bound search engine and overriding the virtual member function `constrain()` of the model. The `constrain()` function is invoked by the search engine whenever a solution is found and takes the most recently found solution as an argument. The idea is to post constraints on the next solution based on the previous solution. These constraints accumulate so that all future solutions will be affected by all already found solutions.

As mentioned, our intention is to replace the current `constrain()` function of Unison with one that instead posts constraints to ensure that future solutions are different in some regard. Remember that in the context of Unison a solution is a valid instruction schedule and allocation of registers.

### 3.2 Cost

An important variable in the Unison model is `cost`. It is the deciding factor of whether a solution is better than another during the branch and bound search. It is a sum of the estimated cost of each basic block, weighted by the estimated execution frequencies [13]. Cost can be either cycles or code size depending on if the optimization goal is speed or size, respectively. That is the `constrain()` function of the Unison model post the constraint that for future combinations to be considered solutions, in addition to the original constraints, the cost must be less than the cost of the previously found solution. Given enough execution time, Unison will thus find the optimal solution with regards to the `cost` variable.

### 3.3 disUnison

The implementation that makes up the following strategies that are used in the experiment is henceforth referred to as disUnison (from the word disunity). It is a variation of the original Unison model, where the key difference is that the behaviour of the branch and bound search is modified. The bulk of the Unison model, that ensures functional code is emitted, is still present in the disUnison model.

For the purposes of this thesis we define three strategies to be evaluated and compared. The goal of each strategy is to provide a popula-
tion of as diverse versions of an executable as possible while incurring as little overhead as possible. The chosen strategies are *enumeration*, *instruction schedule* and *register allocation*, and the motivation for each is presented in each respective subsection.

### 3.3.1 Enumeration

The name of this strategy comes from the fact that we are only concerned that the solutions are different, not how they differ. During search the Gecode search engine will never explore the same combination twice, and thus never generate two equal solutions. The strategy is thus to not post any constraints at all and let the solver generate all possible combinations. We enumerate the solutions.

Unison can differ the solutions in four main ways:

- The order of the operations is different
- Operands are connected to different registers
- Execute a copy using a different instruction (or not at all)
- Split live-ranges and spill temporaries differently

The results of this strategy serve as a baseline for the program. It is literally what happens if we do nothing.

### 3.3.2 Registers

The strategy to diversify the register allocation of the resulting binaries is an attractive one due to causing no run-time overhead. Consequently, if it introduces significant diversity it is an excellent candidate. In addition, as register allocation is one of Unison’s primary purpose it feels like a natural strategy to explore.

There are two variables and one set that are of concern when diversifying the register allocation. Their description from the Unison documentation are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>$P$</td>
<td>The set of all operands in the program</td>
</tr>
<tr>
<td>Variable</td>
<td>$x_p \in {0, 1}$  if operand $p$ is connected (false/true)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$r y_p \in \mathbb{N}_0$</td>
<td>register to which operand $p$ is assigned</td>
</tr>
</tbody>
</table>

In pseudo mathematical notation we want to post the constraint:
CHAPTER 3. DIVERSIFICATION WITH UNISON  

\[ \neg(\bigwedge_{p \in P} ((x_p = \text{prev}.x_p) \land (r_y p = \text{prev}.r_y p))) \]

Very important to note is that the variables preceded by \text{prev}. (i.e. \text{prev}.x_p and \text{prev}.r_y p) represents an actual value. More precisely the value that is part of the previous solution. For example \text{prev}.x_y p, represents the register to which operand p is assigned in the previous solution. The variables not preceded by \text{prev}. are variables in the constraint programming sense and their corresponding domains make up the remaining combinations to explore.

With other words we disallow the exact same combination of connected (used) operands and operand to register mapping.

The \text{r}_y p variable of the model is actually an auxiliary variable that combines the \text{r}_t and \text{y}_p variables. The \text{r}_t variable represents which register temporary t is assigned, and \text{y}_p represents which temporary is connected to operand p. In other words, \text{r}_y p is implemented as \text{r}(\text{y}(p)). This distinction is important during search, and in particular when branching.

### 3.3.3 Instruction Schedule

Given how Unison functions diversifying the instruction schedule is an exciting strategy. As mentioned in Section 2.4 Unison explores optional copies. In practice this means that during pre-processing optional operations are inserted so not only does Unison decide on the order the instructions are executed, but in a limited capacity Unison also inserts instructions (or deems instructions unnecessary). For the purposes of breaking gadgets shifting instructions can help immensely as an adversary is reliant on the exact addresses of the gadgets.

There are two variables and one set of interest for this strategy. In the Unison documentation they are described as follow:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>(O)</td>
<td>The set of all operations in the program</td>
</tr>
<tr>
<td>Variable</td>
<td>(a_o \in {0, 1})</td>
<td>if operation (o) is active (false/true)</td>
</tr>
<tr>
<td></td>
<td>(c_o \in \mathbb{N}_0)</td>
<td>issue cycle of operation (o)</td>
</tr>
</tbody>
</table>

In order to disallow the same set of active operations combined with the same issue cycles we post the constraint:

\[ \neg((\bigwedge_{o \in O} (a_o = \text{prev}.a_o)) \land (\bigwedge_{m \in \{O|\text{prev}.a_m=1\}} (c_m = \text{prev}.c_m))) \]
Just as for the registers strategy constraint, the variables preceded by \textit{prev.} represent the actual value that is part of the previous solution, whereas the ones not preceded by \textit{prev.} are the constraint programming variables used when searching for future solutions.

The constraint described with words is that we disallow the exact same combination of active operations and their corresponding issue cycle. We are only concerned about the issue cycle of the previous solution’s active operations, but we do take into account that future solutions can have the same instructions issued at the same cycles if it has fewer or more active operations.

As mentioned in Section 2.4, some operations can be implemented by multiple instructions. For the purpose of breaking gadgets we do not want to allow functionally equivalent instructions. While they would indeed make the sequence of bytes differ, the functionality would be the same and the gadget would survive.

### 3.4 Branching Strategy

The disUnison models use the same branching strategy as the original Unison model. When the search engine reaches a state where branching is necessary the first decision is to assign the \textit{cost} (see Section 3.2) variable to its lowest possible value. If the \textit{cost} variable is already assigned, the branching is done as follows, in the order listed:

1. assign the active operations. (the $a_o$ variable)
2. assign which instruction should implement each operation.
3. assign which temporary is connected to each operand. (the $y_p$ variable)
4. assign which cycle each operation is issued. (the $c_o$ variable)
5. assign which register is assigned to which temporary. (the $r_t$ variable)

### 3.5 Sampling Rate

When exploring combinations with a constraint solver similar solutions are found close to each-other (in-time). A factor in diversification
might be to exploit this property alongside the diversification strategy. Certain strategies might be favorable when considering execution time but not particularly good at breaking gadgets. However, if e.g. 100 solutions are discarded between every emitted executable perhaps more gadgets are broken.

In the original Unison model the cost variable is used both for branching and during the branch and bound process. As mentioned in Section 3.4, in the disUnison model it is still used for branching. Lower cost combinations are explored first. However, future solutions are not bound to have lower cost. Discarding solutions can thus impact performance in a negative way in the sense that the later versions might have a higher cost, resulting in a wider cost distribution across all program versions.

Sampling rates of 1, 10, 100 and 1000 will be evaluated for every strategy, where a sampling rate of 100 means that every 100th solution is kept. Generating 1000 versions at a sampling rate of 100 would mean that 100000 solutions are explored, 1000 are emitted and 99000 are discarded.

The number of possible combinations is of course not limitless. 1000 versions at a sampling rate of 1000 means that there needs to be at least 1000000 possible solutions. Unison works at the function level and for every function there might not be 1000000 possible versions.

Total number of possible combinations would be an interesting metric to evaluate, unfortunately it varies widely between functions and for some it might require days of search. Empirically, most functions in the suite to be used do have 1000000 versions, so 1000 versions appears to be a good number of versions to generate.

For those function where 1000 versions cannot be generated for the given strategy and sampling rate the ones that have been generated will be re-used so that 1000 program versions can still be generated. More information about these functions can be seen in appendix A.

### 3.6 Target Architecture

Unison does not currently support the x86 or x86-64 architecture. Only ARM, Hexagon and MIPS are supported [14]. None of the supported architectures are generally considered when testing automated software diversity, but for the purposes of evaluating the use of a system-
atic approach the supported architectures offers a glimpse at the potential. For the experiment the code will be compiled for the Hexagon architecture.

Hexagon functions very differently from x86 but for our purposes targeting Hexagon will still hint at the gadget-breaking potential of the systematic approach. After all, breaking gadgets is about shifting, adding, removing or otherwise modifying *any* instruction in the program, and since the diversification strategies (see Section 3.3) are applied globally they are supposedly equally effective regardless of the placement or structure of the instruction.
Chapter 4

Experimental Setup

In order to evaluate both the code generator and the generated code for each strategy a population of programs for each strategy is required. In this section the data set, the evaluation metrics and the process for generating the program populations will be presented.

The experiment will be carried out on a computer running a 4-core Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz and with 8 gigabytes of memory.

4.1 Data Set

The data set to be used is part of the Unison test suite for the Hexagon architecture. In total 23 functions will be used, each of which is from a benchmark in the SPEC2006\(^1\) suite. These function will together make up a program. Since they do not make up a complete executable they cannot be linked nor executed. Linking will instead be simulated by placing them in the same order every time to ensure a fair comparison between strategies and sampling rates.

As mentioned in Section 3.1 Unison works on the function level, and so does disUnison. 1000 versions of each function will be generated and labeled from 0 through 999. Version 0 of each function will make up program version 0, version 1 of each function will make up program version 1 and so forth. That is, one program version consists of 23 functions, and there will be a total of 1000 program versions for each strategy and sampling rate, yielding a total of 12 programs with

\(^{1}\text{https://www.spec.org/cpu2006/ (visited on 21/06/2018)}\)
1000 versions each.

### 4.2 Metrics

The metrics to be evaluated are surviving gadgets, cost (See 3.2), both speed and size, and the execution time of the code generator.

The cost metric is calculated by a tool called *uni analyze*, which is part of the Unison toolchain. It accepts the LLVM MIR of a function as input and outputs the estimated cost for each optimization goal as described in Section 3.2. The cost in terms of size of a program version will be calculated as the sum of the cost for each function version that makes up the program version in question (i.e., the total number of instructions in the program version). The cost in speed of a program version will be calculated as the mean between the cost of the corresponding functions.

For the experiment the optimization goal will be speed, and thus solutions that are estimated to execute faster are generated first. However, both cost in speed and cost in size will be presented in the results section.

Surviving gadgets will be calculated as the ratio of which each gadget appears among the population of program versions. All gadgets present in any of the 1000 program versions will be enumerated. The ratio is calculated as the number of occurrences of every unique gadget divided by the number of versions, which in this case is 1000. In other words a ratio of 100% means that the gadget appears in all version and the strategy was not effective at breaking that gadget. Similarly a very low ratio (close to 0%) would mean that the strategy was effective as the gadget only appears in a small number of the programs.

### 4.3 Generation Process

As mentioned in Section 3.5 most functions in the suite has 1000000 different versions so all strategies will be tested for sampling rates up to 1000. There is a lot of disk I/O involved in generating all versions and even more so when calculating the metrics. Thus, sampling a population of 1000 is empirically a good choice to keep the calculations relatively fast.

Our process to generate our test data is as follows:
1. For every function generate 1000 versions with "speed" as the optimization goal.

2. Version 0 of every function will make up program 0, version 1 will make up program 1 and so forth.

3. Calculate/record our metrics.
   - Execution time of code generator
   - Cost
     - speed
     - size
   - Frequency of surviving gadgets

4. Repeat for all strategies.
   - Enumerate.
   - Registers.
   - Schedule.

5. Repeat for all sampling rates.
   - 1.
   - 10.
   - 100.
   - 1000.

In other words, for every strategy and sampling 1000 versions of each function will be generated. Each combination of strategy and sampling rate will thus have 23000 function versions associated with it. Version 0 of every function for each strategy and sampling rate will make up program version 0 for the corresponding strategy and sampling rate combination. Version 1 will make up program version 1 and so forth. Each program version will consist of 23 functions so the process will yield 1000 program versions for each combination of strategy and sampling rate.

The resulting test data will be 12 programs, one for each combination of strategy and sampling rate. Each of these 12 programs represents a population of 1000 different versions, for a total of 12000
program versions. Each version consists of 23 functions for a total of 276000 function versions. The result will emphasize comparison between the sampling rates for each strategy as well as the difference between the strategies.
Chapter 5

Results and Discussion

In this chapter each metric mentioned in Section 4.2 will be presented in its own section. An interpretation of the result will also be presented for each metric, but all general discussion will be in chapter 6.

5.1 Generation Time

The execution times of the constraint solver at sampling rate 1000 are shown in Figure 5.1. Each marker represents the completed generation of a function. That is, for each marker 1000000 (one million) solutions have been explored. For a few functions fewer than 1000 solutions were found (for a complete breakdown see appendix A).

While the total execution time is daunting, it does represent about 23 million solutions found (but only 23000 emitted). The enumerate strategy is fairly quick: 51 minutes for one million solutions of every function means that on average, one solution was found every 139ns. The same number for registers and schedule is 343ns and 1095ns respectively. To mitigate the impact of a long execution time the solutions can be emitted directly when found.

disUnison does not use the same search heuristics as the base Unison model. The disUnison search heuristics makes implementation easier at the potential cost of execution time. The problem with optimizing the branching and search heuristics of the model is that the performance impact will largely have to be determined empirically. Supposedly, the optimal search heuristic for the registers strategy would not be the same as the optimal search heuristic for the schedule strategy.
Figure 5.1: The execution time of the code generator at sampling rate 1000 (i.e. 1000000 solutions for every function). Each marker represents the finished generation of a function. The markers are ordered so that the nth marker on every line represents the same function. Total time is annotated as hours and minutes.

5.2 Cost

Figure 5.2 and Figure 5.3 shows the distributions of the estimated cost in cycles and the code size for each strategy and sampling rate respectively. The cost in terms of code size is calculated as the sum of the cost for each function, i.e. the number of instructions in the program version. Cost in terms of speed is calculated as the mean between the costs of the individual functions that make up the program version. That is, for every strategy and sampling rate there are 1000 values for each cost metric. For the size metric each of those values is a sum of 23 values. For the speed metric each of those values is a mean between 23 values.
Figure 5.2: The cost (speed) distributions for every strategy and sampling rate. The cost of the LLVM solution is included for reference.

All strategies perform better for lower sampling rates both in terms of speed and in terms of code size. As described in section 3.5, this is expected. Both the enumerate and registers strategy performs very well compared to the LLVM solution in terms of speed. The schedule strategy incurs a slight overhead for the lowest sampling rate and a significant overhead for the three higher sampling rates.

The fact that no solution has a size lower than the LLVM solution can be explained by referring to the optimization goal used, which was speed. The code size has not been taken into account during branching and no attempt has been made to optimize it.

Interesting to note is that all strategies and sampling rates have found a solution with an equally low speed. This is the very first solution found and it is upon this solution the first strategy related constraints are based. Thanks to the search heuristic, where lower cost solutions are explored first, this is also the optimal speed.
Figure 5.3: The code size distributions for every strategy and sampling rate. The cost of the LLVM solution is included for reference.

5.3 Surviving Gadgets

Figure 5.4 shows the occurrence ratio of each gadget for the different strategies and sampling rates. The x-axis shows a gadget id and there is no definitive correlation between the gadgets across strategies and sampling rates. The ids are ordered from most frequent to least frequent to better facilitate comparison. The first non-zero x-tick refers to the id of the first gadget that only appear once and the second non-zero x-tick is the total number of unique gadgets. E.g the enumerate strategy at sampling rate 1 has $1561 - 906 = 655$ gadgets that only appear in one program version. Strategies and sampling rates that were more effective have a higher number of unique gadgets. The y-axis is set to a logarithmic scale since the bulk of the data is below 1%. The frequency of the most frequent gadget is annotated.

As seen in Figure 5.4, neither the enumeration nor the registers
Figure 5.4: The ratio of occurrence for each gadget broken down by strategy and sampling rate. The x-axis displays the gadget ids. The data is sorted from most to least frequent to allow for a better overview.
strategy were particularly effective at breaking gadgets for any sampling rate, when compared to the schedule strategy. There is a slight improvement for higher sampling rates but it is not particularly impressive even at sampling rate 1000. There are still many gadgets that survives across a large percentage of versions. The schedule strategy, however, performed well with no gadget being present in even 50% of all versions for the lowest sampling rate and for sampling rate 1000 about 82% of gadgets only appear in one version.

If an adversary were to construct a payload for a binary from the schedule strategy at sampling rate 1000 the chances of it working on another binary in that population is very small. Even if the payload only relies on a single gadget, and that gadget happens to be the most frequent one, the payload would only work on about 24% of the binaries. It is, however, much more likely that an adversary needs multiple gadgets. Since 82% of all gadgets are unique to their program version and the gadgets has to be exactly equal on for the payload to work on another binary it is very unlikely that the adversary can re-use the payload at all. Unfortunately the schedule strategy also incurred the most overhead, as seen in Section 5.2.

The difference in behavior of the strategies is discussed in Section 6.1.
Chapter 6

Discussion

6.1 Discussion

The core of the disUnison model, the three strategies, are implemented in just a handful lines of code and, thanks to being an extension of the Unison model, there is never a risk of breaking the functionality of the generated code. This is not to say that every diversification strategy allows for proper executable code to be generated, but thanks to the nature of constraint solving the result will either be all possible solutions (given enough execution time) or a proof that no solution, and thus no proper executable, exists.

Apart from cost, the set of active operations is the first variable to be assigned during search (see Section 3.4). This partly explains the long execution time of the code generator for the schedule strategy. When the issue cycles of a set of active operations have been exhausted all branching decisions following the cost are discarded. In contrast the registers strategy can keep the decisions relating to active operations and which instruction implements which operation and the enumerate strategy can keep all but the very last decision taken.

As an explanation for the effectiveness of the schedule strategy consider what was mentioned in Section 3.3.3. When applying the schedule strategy, not only can the order of operations change between solutions, the number of operations can also change. During search when applying the schedule strategy all combinations of issue cycles for a valid set of active operations will be explored. When all possible orderings of that particular set has been exhausted the search engine will attempt to find a new set of active operations. This new set will then
contain more or less operations than the previous set. By introducing or removing instructions either all following instructions are effectively shifted. Since code re-use attacks rely on the exact address of the instructions to be executed, a simple shift can foil the attack. In addition to breaking gadgets by simply swapping independent instructions, these shifts are likely the cause of the effectiveness of the schedule strategy. However, inserting instructions will in most cases mean a larger overhead.

There is a constant trade-off between diversity and execution overhead when generating diverse populations of executables and the systematic approach is not excluded from this. From the results we can deduce that there is a correlation that a more diverse population in terms of gadgets incurs a wider distribution and a higher mean for the cost metric for the implemented strategies. However, when comparing sampling rates 10 and 100 of the enumerate and registers strategy the higher sampling rate generates a noticeably more diverse population, but the mean cost in speed is virtually unchanged and the code size is only increased by a handful of percentage points. Perhaps a more advanced strategy can lessen this gap even more and still provide a diverse population of executables.

### 6.2 Threats to Validity

Homescu et al. [8] found 433.milc from the SPEC2006 benchmark suite to be representative in terms of surviving gadgets. Unfortunately 433.milc was too large for the experiment and the Unison test suite had to be used instead. Most of the functions in 433.milc were too large to find even one solution (one searching for about one hour). This is an obvious problem for any practical purpose of the systematic approach. One solution to this problem is to modify the search heuristics of the disUnison model even further. Unfortunately it would be difficult, if not impossible, to find an optimal, generally-applicable search strategy.

For a more proper comparison of the systematic approach tests would have to be repeated for a more comprehensive data set and target the x86 or x86-64 architecture. As mentioned in Section 3.6 Unison in its current state does not support the x86 and x86-64 architecture and the tests were conducted for the Hexagon architecture. This is, of
course, not optimal but given that the strategies are not explicitly tied to the underlying instructions and that they are potentially applied to all instructions the results are still promising. If or when Unison or a similar tool implements support for the x86 and x86-64 architecture, the experiment would have been repeated on to test whether or not the strategies are equally performant.

6.3 Future Work

In addition to addressing the shortcomings by targeting the x86 platform and improving the search heuristics to be able to compile larger functions there is much to explore.

Regardless of what the selection of strategies may indicate the possibilities for diversification are far broader than when approaching the problem in terms of register allocation and instruction scheduling as separate procedures. It is important to keep in mind that more unorthodox strategies that exploit the combined approach might be even more performant. As mentioned in 3.1 the Unison model consists of more variables than the 4 explored in this experiment, all of which offers potential for diversity. The strategies can also be combined, both in terms of combining the constraints as well as combining the resulting function differently. E.g one could construct a program version by combining function versions from different strategies and sampling rates.

Unison (and disUnison) accepts the basic LLVM-solution as an optional parameter, and can post constraints to only generate solutions that are better. In other words, we can generate executables with zero overhead with respect to LLVM’s solution. Certain strategies would of course have an overhead, but with respect to the optimal solution. There is also the opportunity to limit the resulting population in other ways, e.g only incur a 5% overhead compared to the LLVM solution. By not randomizing we have full control of the process and can limit the resulting code in whatever way is appropriate. As this would limit the number of possible version this optional parameter was not used during the experiment. It is however an exciting factor to consider for future work.
6.4 Conclusion

Using a constraint solver to generate diverse binaries is an attractive approach given the ease of implementation and the quality of the generated code. The resulting population of diversified programs shows that the systematic approach has great potential at breaking gadgets as well as providing great control of the incurred overhead. However, the shortcomings of the experiment and the tool are a testament to the future work required. The hypothesis remains unanswered.
Bibliography


## Appendix A

### Functions

#### A.1 Functions and Statistics

Number of solutions found for each function, strategy and sampling rate.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Function</th>
<th>Strategy</th>
<th>Sampling Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Unison</td>
<td>cond-transfer</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>Unison</td>
<td>cond-transfer</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>Unison</td>
<td>cond-transfer</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>alias.get_frame_alias_set</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>alias.get_frame_alias_set</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>alias.get_frame_alias_set</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>c-decl.pushdecl_top_level</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>c-decl.pushdecl_top_level</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>c-decl.pushdecl_top_level</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>expmed.ceil_log2</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>expmed.ceil_log2</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>expmed.ceil_log2</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>insn-output.output_51</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>insn-output.output_51</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>insn-output.output_51</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>jump.unsinged_condition</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>jump.unsinged_condition</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>jump.unsinged_condition</td>
<td>schedule</td>
<td>243 23 3 1</td>
</tr>
<tr>
<td>gcc</td>
<td>rtlannal.insn_dependent_p_1</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>rtlannal.insn_dependent_p_1</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>rtlannal.insn_dependent_p_1</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>rtlannal.parms_set</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>rtlannal.parms_set</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>rtlannal.parms_set</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>varasm.data_section</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>varasm.data_section</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>varasm.data_section</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>gobmk</td>
<td>barriers.autohelperbarrierspat145</td>
<td>enumerate</td>
<td></td>
</tr>
<tr>
<td>gobmk</td>
<td>barriers.autohelperbarrierspat145</td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>gobmk</td>
<td>barriers.autohelperbarrierspat145</td>
<td>schedule</td>
<td></td>
</tr>
<tr>
<td>Benchmark</td>
<td>Function</td>
<td>Strategy</td>
<td>Sampling Rate</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------</td>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>gobmk</td>
<td>board.get_last_player</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>board.get_last_player</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>board.get_last_player</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>792</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_attackpat.autohelperowl_attackpat68</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_attackpat.autohelperowl_attackpat68</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_attackpat.autohelperowl_attackpat68</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>792</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_defendpat.autohelperowl_defendpat421</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_defendpat.autohelperowl_defendpat421</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_defendpat.autohelperowl_defendpat421</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_vital_apat.autohelperowl_vital_apat34</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_vital_apat.autohelperowl_vital_apat34</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>owl_vital_apat.autohelperowl_vital_apat34</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>patterns.autohelperpat1088</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>patterns.autohelperpat1088</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>gobmk</td>
<td>patterns.autohelperpat1088</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>h264ref</td>
<td>memalloc.no_mem_exit</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>h264ref</td>
<td>memalloc.no_mem_exit</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>h264ref</td>
<td>memalloc.no_mem_exit</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>h264ref</td>
<td>sei.UpdateRandomAccess</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>h264ref</td>
<td>sei.UpdateRandomAccess</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>h264ref</td>
<td>sei.UpdateRandomAccess</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>mesa.api</td>
<td>glIndexd</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>mesa.api</td>
<td>glIndexd</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>mesa.api</td>
<td>glIndexd</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>sphinx3</td>
<td>glist.glist_tail</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>sphinx3</td>
<td>glist.glist_tail</td>
<td>registers</td>
<td>1000</td>
</tr>
<tr>
<td>sphinx3</td>
<td>glist.glist_tail</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>sphinx3</td>
<td>profile.ptmr_init</td>
<td>enumerate</td>
<td>1000</td>
</tr>
<tr>
<td>sphinx3</td>
<td>profile.ptmr_init</td>
<td>registers</td>
<td>17</td>
</tr>
<tr>
<td>sphinx3</td>
<td>profile.ptmr_init</td>
<td>schedule</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>hmmer</td>
<td>tophits.AllocFancyAli</td>
<td>all</td>
<td>0</td>
</tr>
<tr>
<td>gcc</td>
<td>xexit.xexit</td>
<td>all</td>
<td>0</td>
</tr>
</tbody>
</table>

There were no solutions found for the functions hmmer.tophits.AllocFancyAli and gcc.xexit.xexit. Therefore these were excluded in the experiment. As they are part of the Unison test suite they are included here for completeness.
Appendix B

ROP Exploit

Python code that executes a ROP exploit for the ROP emporium 64-bit split challenge.\(^1\). Requires the pwntools package\(^2\) and that the challenge files are located in the same directory as the script.

```python
#!/bin/python2
from pwn import *

elf = context.binary = ELF('split')

context.log_level = 'debug'

cat_addr = 0x601060
system = 0x400810
gadget = 0x400883
info("target string: %#x", cat_addr)
info("address of system(): %#x", system)
info("gadget: %#x", gadget)

rop_chain = p64(gadget, endian='little')
rop_chain += p64(cat_addr, endian='little')
rop_chain += p64(system, endian='little')

payload = "A"*40 + rop_chain
print(payload.encode('hex'))
```

\(^1\)https://ropemporium.com/challenge/split.html (visited on 20/06/2018)

\(^2\)https://github.com/Gallopsled/pwntools (visited on 20/06/2018)
io = process(elf.path)
io.sendline(payload)
io.wait_for_close()
io.recv()