Detecting Behaviorally Equivalent Functions via Symbolic Execution

Kesha Hietala

Submitted under the supervision of Stephen McCamant to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science, *summa cum laude* in Computer Science.

2016

Acknowledgments

I would like to thank my thesis advisor, Stephen McCamant, and thesis readers, Eric Van Wyk and Gopalan Nadathur, for their guidance and feedback in improving this thesis. I would also like to thank Vaibhav Sharma for all his work on our adaptor synthesis tool.

Abstract

Software bugs are a reality of programming. They can be difficult to identify and resolve, even for the most experienced programmers. Certain bugs may even be impossible to remove because they provide some desired functionality. For this reason, designers of modern security-critical applications must accept the inevitable existence of bugs and find ways to detect and recover from the errors they cause. One approach to error detection involves running multiple implementations of a single program at the same time, on the same input, and comparing the results. Divergence of the behavior of the different implementations indicates the existence of a bug.

The question we consider in this paper is how to construct these diverse implementations of security-critical programs in a cost-effective way. The solution we propose is to first find existing diverse function implementations and then use these function implementations as building blocks for diverse program implementations. To find diverse function implementations, we use a technique we call *adaptor synthesis* to compare arbitrary functions for behavioral equivalence. To account for differences in input argument structure between arbitrary functions we allow for adaptor functions, or *adaptors*, that convert from one argument structure to another. Using adaptors, the problem of determining whether two arbitrary functions are behaviorally equivalent becomes the problem of synthesizing an adaptor between the two functions that makes their output equivalent on all inputs in a specified domain.

Along with presenting our adaptor synthesis technique, we describe an implementation for comparing functions for behavioral equivalence at the binary level on the Linux x86-64 platform using a family of adaptors that allows arithmetic combinations of integer values.

Contents

Chapter 1: Introduction

1.1 Objective

The purpose of this work is to provide an automated method for finding behaviorally equivalent functions, where *behavioral equivalence* relates to input and output behavior. To do this, we use a technique we call *adaptor synthesis* to search for adaptor functions that transform a function's input, causing its behavior to match that of a specification function. The intended application of this work is in constructing different program implementations for a multivariant execution system. In this paper we present both a general approach for finding adaptor functions and an implementation of our approach where adaptor functions are constructed from arithmetic operations on integer values.

1.2 Background

This section provides a brief introduction to two areas of research central to our work: *software diversity* and *program synthesis*. A more comprehensive description of the current state of relevant research in both fields can be found in Chapter 2. This section also provides a quick introduction to our adaptor synthesis technique.

1.2.1 Software Diversity

Security-oriented software diversity aims to introduce implementation uncertainty into security-sensitive programs. The intention is that this introduced uncertainty will force attackers to make guesses about some program details, greatly increasing the cost of developing certain types of attacks. For example, address space layout randomization (ASLR), a technique implemented on most modern operating systems, provides protection against buffer overflow attacks by randomizing the location where code and data are loaded into memory. Similar randomization can also provide probabilistic protection against a variety of other memory errors including dangling pointers and uninitialized reads in unsafe languages like C and C++ [4].

However, in today's computers, diversity is the exception rather than the rule. Hardware, operating systems, and applications are all highly standardized to allow for easy distribution and maintenance. The significance of this from a security standpoint is that once an attacker chooses a target system, he or she can easily download an identical copy of the operating system and commodity software running on that target and probe them for vulnerabilities. Any exploit that the attacker finds can then be used not only against the intended target, but also against any system running that particular operating system/vulnerable software combination. The 2014 Heartbleed vulnerability in OpenSSL, which allowed attackers to remotely read protected data from an estimated 24-55% of popular HTTPS sites [9], is one example of a widespread attack made possible by the lack of diversity in modern critical software. Research into software diversity aims to make exploiting software vulnerabilities more expensive by randomizing implementation details of programs across different systems. This randomization makes it more difficult both for an attacker to obtain an exact copy of the software on their target system and to carry out a large-scale attack with a single exploit.

This project focuses on an application of software diversity known as N-version programming. In N-version programming, the goal is to obtain fault tolerance by executing diverse implementations of a program at the same time, on the same input, and then comparing the implementations' results. Traditionally, these different implementations (or 'versions') are developed separately by different teams of programmers using different programming languages and algorithms. Under the assumption that independently developed pieces of software have independent bugs— making it statistically unlikely that many versions will have the same bug— an error in one program version can be detected when that version's behavior begins to deviate from the behavior of the others. N-version programming systems can also recover from some errors by choosing the majority result of a computation as the 'correct' result. This means that in order for an attack on an N-version system to be effective, an attacker must compromise a majority of the versions at the same time with the same input.

But there are two major drawbacks to N-version programming: 1) it requires manual development and maintenance of multiple implementations of the same functionality, which can become costly, and 2) it is difficult to reason about the independence of bugs among the different versions and to find what bugs they have in common. Recent work on N-variant systems has aimed to address these drawbacks by automating the construction of 'variants' that have certain desirable security properties, such as having disjoint vulnerabilities. (Note that the distinction between a 'variant' and a 'version' is that variants are automatically constructed, while versions are manually developed.) However, current N-variant approaches are limited in the types of diversity that they can introduce. For example, they cannot diversify high-level implementation details like choice of algorithms or data structures. We are interested in whether there is a way to construct more diverse program variants using mined behaviorally equivalent functions. This paper focuses primarily on the task of finding behaviorally equivalent functions.

Notice that we focus on *finding* existing behaviorally equivalent functions rather than constructing diversified implementations of a given functionality. We suggest that this is possible because of our observation that in any large codebase there are likely several different functions that effectively 'do the same thing.' This could be the result of a variety of factors including poor communication between collaborating developers, incomplete code documentation, or a lack of understanding of the capabilities of existing functions— all of which lead to developers unintentionally re-implementing existing code. By reusing existing function implementations we hope to avoid the programming effort required by a traditional N-version approach, while still allowing for more interesting diversity than is in a typical N-

variant approach. But note that our work does not address the second drawback mentioned above— we cannot guarantee that the different function implementations we find actually have independent bugs.

1.2.2 Program Synthesis

In order to find equivalent function implementations, we rely on a technique known as *program synthesis*. The motivating principle behind program synthesis is that the most productive way for people to write programs is for them to specify *what* they want a program to do and have the computer decide *how* to do it. When programmers can specify computation at a high level, they are able to devote more of their time to creative problem solving and spend less of their time on obscure implementation details like array index expressions and bit manipulations.

This project focuses on a type of program synthesis known as counterexample-guided inductive synthesis (CEGIS). The general CEGIS approach, depicted in Figure 1.1, begins with a specification of the desired program and then alternates between using a *synthesizer* and a *verifier* until a program satisfying that specification is produced. The job of the synthesizer is to produce a candidate program that might satisfy the specification, while the job of the verifier is to decide whether that candidate program actually satisfies the specification. If not, the verifier provides feedback to the synthesizer to help it search for a new candidate. This feedback is typically a 'counterexample,' which is an input on which the candidate program failed to satisfy the specification.

We use CEGIS when comparing functions for behavioral equivalence. Roughly speaking, we consider two functions to be behaviorally equivalent if they 'do the same thing,' meaning that they produce equivalent output on most inputs. We require equivalent output on *most* rather than *all* inputs because we want to allow for equivalence between functions that have different behaviors on 'bad' inputs (e.g. inputs that trigger a bug or error handling). This

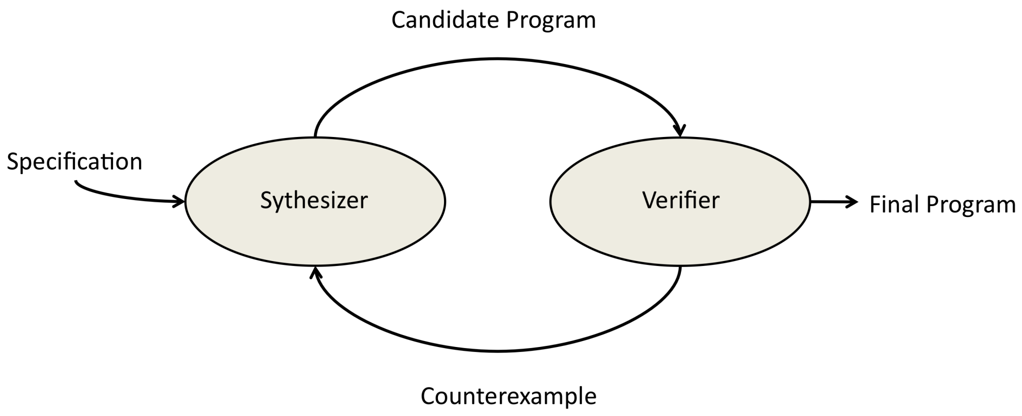


Figure 1.1: Diagram of the standard CEGIS loop.

enables the construction of program variants with different bugs.

Because we are targeting functions that have been independently re-implemented, we want to allow for equivalence between functions with different argument structures. To make the argument structure of one function match that of another we use CEGIS to synthesize an adaptor function (or ‘adaptor’). This approach is described in more detail in Chapter 3, but the general idea is to compare functions f_1 and f_2 for equivalence using a CEGIS loop where the specification is provided by the input and output behavior of f_1, the synthesizer searches for adaptors over the arguments of f_2 that make the behavior of f_2 match that of f_1, and the verifier looks for inputs that cause f_1 and f_2 to have different behaviors with a given adaptor.

Chapter 2: Literature Review

In this chapter we describe the current state of relevant research in N-version and N-variant systems, detection of equivalent code, and counterexample-guided inductive synthesis.

2.1 N-version and N-variant Systems

The technique of N-version programming, as described in Section 1.2.1, was introduced as early as the 1970s [2]. Since that time, a variety of work has analyzed the assumption of the independence of errors among different versions [24], the effectiveness of N-version programming in the presence of overlapping errors [10], and the practicality from a software engineering standpoint of developing multiple (potentially less-reliable) versions as opposed to a single highly-reliable version [17]. Generally this work concludes that running different program versions simultaneously and using a voting mechanism can produce a system that is more reliable than any of the versions individually, and that developing different program versions is a good way to achieve fault tolerance, especially considering that it is difficult to develop one really good, bug-free version. However, this work also suggests that it is difficult to make formal guarantees about the security that an N-version system provides.

Recent approaches to N-version programming have taken advantage of *natural diversity* in related software products [14, 34]. Natural diversity refers to diverse implementations of the same functionality that emerge naturally, often as the result of economic competition. Natural diversity can be seen in the diverse implementations of, but similar functionalities provided by, web browsers, operating systems, firewalls, database management systems, and so on [3]. These naturally-emerging diverse implementations of commercial-of-the-shelf (COTS) products are good candidates for N-version programming because they tend to have non-overlapping bugs [15, 16]. However, determining which COTS products provide the same functionality (and so can be used together in an N-version system) is still a manual process. Our work aims to automatically detect natural diversity when searching for behaviorally equivalent functions.

Another recent take on the idea of executing multiple implementations of a program simultaneously for fault tolerance is called N-variant (or multivariant) execution [7, 31]. The distinction between a 'version' and a 'variant' is that a version is manually developed while a variant is automatically generated. Cox et al. [7] show how to construct variants in such a way that they have disjoint vulnerabilities with respect to certain classes of attacks. They focus on two diversification techniques: memory address partitioning, which provides protection against attacks involving absolute memory addresses, and instruction tagging, which detects attempts to execute injected code. Salamat et al. [31] introduce a user-space multivariant execution environment (MVEE) that monitors multiple variants of a program running in parallel and show that this technique is effective in detecting and preventing code injection attacks. MVEE variants are generated using different directions of stack growth and system call number randomization. One restriction on the MVEE is that it requires all variants to make the same system calls, in the same order, with the same arguments. This restriction motivates some of our handling of side-effects when comparing functions for equivalence.

N-variant execution is attractive because it allows for automated construction of different program implementations and enables formal arguments about a system's security. However, existing variant construction techniques are limited in the types of diversity that they can introduce. Techniques like memory address partitioning, instruction tagging, memory rearrangement, and system call randomization all preserve program control flow and semantics, which means that they cannot detect common attacks such as cross-site scripting, directory traversal, and SQL injection that take advantage of flaws in the program logic (i.e. program semantics) itself [5]. Our approach of using diverse function implementations to construct program variants has the potential to detect a broad range of interesting attacks.

We envision that the diverse function versions that our tool discovers can be used in an automated way to construct program variants. A discussion of how to construct these program variants is beyond the scope of this paper, but the general idea is to replace (statically or dynamically) function calls in a source program with different, but equivalent, function calls to produce a new variant of that program [5].

Another common approach to developing variants, which is not obviously compatible with our idea of using behaviorally equivalent functions, relies on compiler-based randomization [25]. It is often convenient to modify a compiler to support randomization because compilers already perform many of the analyses required for randomization and are designed to target many different architectures. However, compiler-based variant generation requires that the source code of the program to be randomized is available and that it is possible to customize the compiler. Our approach to constructing variants does not rely on the availability of source code and is compatible with proprietary compilers. Compiler-based approaches to diversity are also limited in the types of diversity that they can introduce.

2.2 Detecting Equivalent Code

Detecting pieces of equivalent code is useful for many applications including refactoring, code understanding, bug finding, and optimization. The majority of previous work in this area has focused on detecting *syntactically* equivalent code, or 'clones,' which are typically the result of copy-and-paste [20, 23, 26]. Generally speaking, we are not interested in finding syntactically similar functions because such functions were likely not independently implemented and may have similar, if not identical, vulnerabilities.

Recent work has begun to consider the problem of detecting *behaviorally* equivalent pieces of code, or 'semantic clones'. Applications of detecting semantic clones (aside from our motivating application of multivariant execution) include functionality-based refactoring, semantic-aware code search, and checking 'yesterday's code against today's.' Equivalence between code fragments may be defined in terms of internal behavior (e.g. code fragments may be considered equivalent if they have isomorphic program dependence graphs [12]) or input/output behavior. We focus on the latter approach because it allows for equivalence between more computationally diverse code.

Recent tools for detecting semantic clones using an input/output based definition of equivalence include EQMINER from Jiang et al. [21] and UC-KLEE from Ramos and Engler [30]. EQMINER detects behaviorally equivalent code fragments using random testing, and has been used to show that Linux kernel 2.6.24 contains many behaviorally equivalent code fragments that are syntactically different. Similar to our work, EQMINER allows for equivalence between code fragments with different argument structures. Specifically, inputs to a code fragment may be permuted or combined into structs. We view these operations as limited types of adaptor functions between code fragments. UC-KLEE checks for equivalence between arbitrary C functions using symbolic execution, and has been used to detect discrepancies between function implementations in different versions of the C library. UC-KLEE does not support equivalence between functions with significantly different argument structures.

2.3 CEGIS

Program synthesis is an active area of research that has many applications including generating optimal instruction sequences [29, 22], automating repetitive programming, filling in low-level program details after programmer intent has been expressed [33], and even binary diversification [18]. Programs can be synthesized from formal specifications [27], simpler (likely less efficient) programs that have the desired behavior [29, 33, 22], or input/output oracles [19]. We take the second approach to specification in our work, treating existing functions as specifications when synthesizing adaptors.

As described in Section 1.2.2, we focus on a type of synthesis known as counterexample-guided inductive synthesis (CEGIS). CEGIS is useful for our work because it is guaranteed to terminate when the space of possible programs is finite, and if it terminates by producing a program, then that program is guaranteed to be correct with respect to the specification [33]. This means that when comparing functions for equivalence, if we restrict our attention to a finite family of adaptors, then the adaptor synthesis process is guaranteed to terminate with an adaptor that makes the two functions behaviorally equivalent (possibly over a restricted input domain), or some indication that the functions are not behaviorally equivalent with any adaptor in the specified family. Also, although the space of candidate programs may be very large, the search for a correct program under CEGIS tends to take relatively few iterations [33].

Chapter 3: Approach

3.1 Tools

FuzzBALL [28] is a symbolic execution tool for exploring execution paths in binary code. The idea of symbolic execution is to execute programs with certain concrete program state values replaced by *symbolic variables* that keep track of how that state is updated. Using symbolic execution, we can build up *symbolic formulas* that describe a program's output on many different concrete inputs. In the context of our project, symbolic execution allows us to ask questions such as 'what inputs will cause this function to have a different output from the specification?' and 'what program state values will cause this function to have the same output as the specification on a given input?'. FuzzBALL is the backbone of the verifier and synthesizer components of our tool for finding behaviorally equivalent functions.

To check whether constructed symbolic formulas are feasible, FuzzBALL relies on a Satisfiability Modulo Theories (SMT) solver. SMT solvers check the satisfiability of logical formulas over one or more theories. They have become increasingly popular with symbolic execution tools like FuzzBALL because of their natural application to checking conditions on transitions between different program states [8]. For this work we use the STP SMT solver [13].

3.2 Adaptor Synthesis

In this section we describe the idea behind, and our implementation of, what we call *adaptor synthesis*. The goal of adaptor synthesis is to construct an adaptor function, or *adaptor*, that makes the behavior of one function, f_2, match the behavior of another (oracle) function f_1. That is, when the adaptor is applied to the arguments of f_2, f_2 must produce output equivalent to that of f_1 on all inputs (possibly in a restricted domain). Given a correct adaptor over the arguments of f_2, we can use f_2 interchangeably with f_1 at any point in a program. We then say that f_1 and f_2 are *behaviorally equivalent*. We do not require the existence of an adaptor in the other direction (i.e. an adaptor over the arguments of f_1) in order for f_1 and f_2 to be considered equivalent. In the case where no adaptor over the arguments of f_1 or f_2 makes the behavior of f_1 and f_2 match, we say that the functions are not behaviorally equivalent. It is possible for there to be multiple correct adaptors between f_1 and f_2.

Our ability to determine whether two functions are behaviorally equivalent depends on the types of adaptors that we allow. More restrictive adaptors allow for equivalence between fewer functions. More expressive adaptors allow for equivalence between more functions, which helps in discovering diverse implementations of a functionality, but increases the search space for synthesis, often resulting in significant performance impacts.

Our approach to synthesizing adaptors can be summarized as follows: given two functions, f_1 and f_2, that take arguments $(x_1, ..., x_n) \in X$ and $(y_1, ..., y_m) \in Y$ respectively, and a finite family of adaptor functions A, our goal is to find a function $a \in A$, $a : X \mapsto Y$ such that $f_1(x_1, ..., x_n) = f_2(a(x_1, ..., x_n))$ for all $(x_1, ..., x_n)$. We do this by alternating between the use of a synthesizer that generates candidate adaptors a and a verifier that looks for values of $x_1, ..., x_n$ that do not satisfy the relation $f_1(x_1, ..., x_n) = f_2(a(x_1, ..., x_n))$ for a given candidate adaptor. This approach, which we refer to as adaptor synthesis, is depicted in Figure 3.1.

This description warrants additional explanation on a few points:

- When comparing the output of f_1 and f_2, we need to take into account both the return values of the functions as well as their side-effects. A function's side-effects include its system calls and modifications it makes to global memory or data pointed to by input pointer arguments.[1]

- Comparing the output of f_1 and f_2 for equivalence is not necessarily the same as checking for equality. For example, if f_1 and f_2 both return zero to indicate 'false' and an arbitrary nonzero value to indicate 'true,' then their return values may be semantically equivalent on an input without being exactly equal. Similar issues arise when comparing side-effects for equivalence because a single system call may have equivalent behavior with different arguments (e.g. two calls to *open* with different buffers containing the same file path), different system calls may have equivalent behavior with certain arguments (e.g. *creat* is equivalent to *open* when its second argument is set to $O\_CREAT|O\_WRONLY|O\_TRUNC$), and writes to different locations in memory may be equivalent if the data written is the same.

- Our synthesis tool relies on the assumption that a function's behavior depends only on its input arguments and not on the environment. To account for effects of the environment on a function's behavior, we could treat environment variables as additional inputs to the functions under consideration.

- The input spaces X and Y of f_1 and f_2 above do not necessarily consist of all possible inputs to f_1 and f_2. For example, say that f_1 takes one argument, and that argument is used in the body of f_1 as an ASCII character. Then the input space X may only consist of integer values between 0 and 127. f_1 may have even been implemented in such a way that its behavior on inputs outside the range 0–127 is undefined. In this situation, it only makes sense to consider the behavior of f_1 on inputs in the range 0–127 when comparing it for equivalence with other functions. Another reason for considering a function's behavior on some (rather than all) inputs comes from our motivating application of multivariant execution. If we can find an adaptor that makes f_1 and f_2 produce the same output on *all* inputs, then f_1 and f_2 have identical bugs, and are not useful for multivariant execution. By ignoring behavior on some inputs, we allow for equivalence between diverse function implementations that have, for example,

[1]The details of handling side-effects are not discussed in this paper because the relevant work was not specific to the author, and the example functions we present here do not have side-effects. However, the current implementation of our tool does have basic handling for side-effects [32]. We require that equivalent functions make the same sequence of system calls with the same arguments (as inspired by the MVEE described by Salamat et al. [31]), and make the same writes to non-local addresses.

different error handling.

- Our definition of equivalence is based only on input and output behavior, and does not take into account intermediate program states. This allows us to detect diverse implementations of the same behavior.

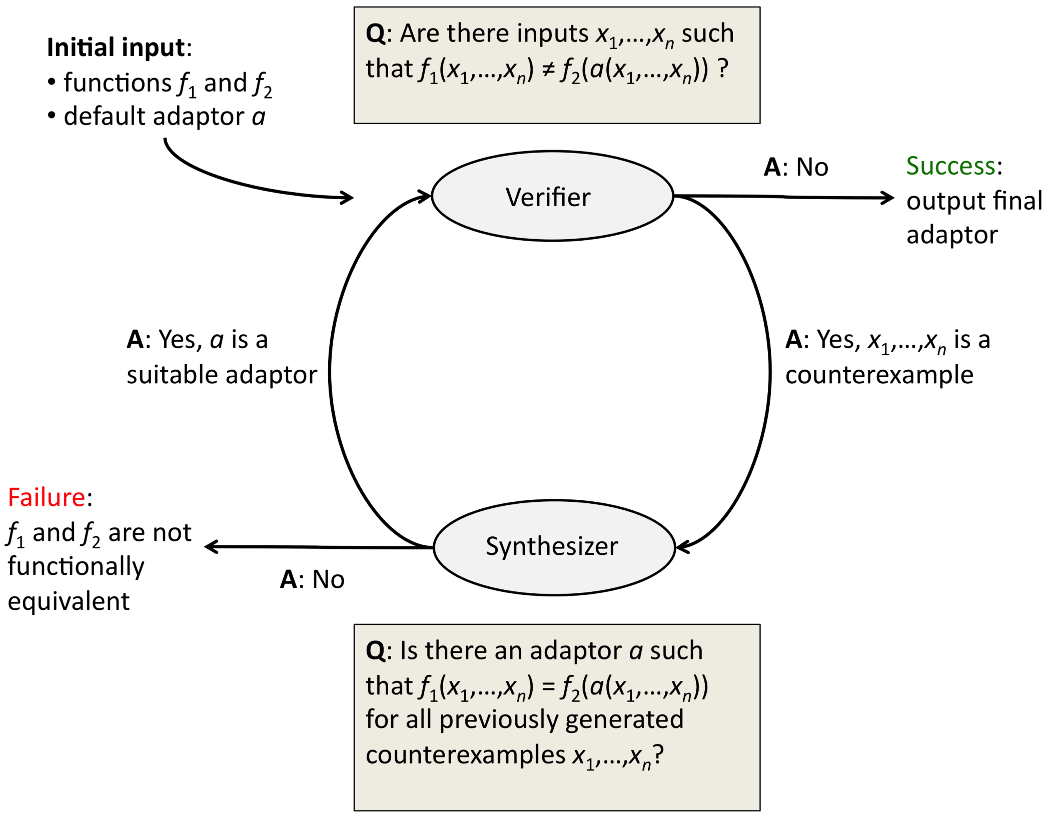

Initial input:
• functions f_1 and f_2
• default adaptor a
Q: Are there inputs $x_1,\ldots,x_n$ such that $f_1(x_1,\ldots,x_n) \neq f_2(a(x_1,\ldots,x_n))$?
Verifier
A: No
Success: output final adaptor
A: Yes, a is a suitable adaptor
A: Yes, $x_1,\ldots,x_n$ is a counterexample
Failure: f_1 and f_2 are not functionally equivalent
Synthesizer
A: No
Q: Is there an adaptor a such that $f_1(x_1,\ldots,x_n) = f_2(a(x_1,\ldots,x_n))$ for all previously generated counterexamples $x_1,\ldots,x_n$?

Figure 3.1: An overview of the CEGIS loop used in adaptor synthesis.

In Chapter 4, we present a family of adaptors that map between integer values using integer arithmetic operations.[2]

To implement the discussed approach, we rely on FuzzBALL's ability to symbolically explore execution paths and return satisfying assignments to symbolic variables on each

[2]We actually developed several types of adaptors as part of this project including adaptors allowing type conversions between arguments, floating point arithmetic operations, and operations on strings. We also added support for applying adaptors to function return values [32]. However, only work specific to the author is described here.

path it explores. During synthesis, we ask FuzzBALL for satisfying assignments on execution paths where the behavior of f_2 matches that of f_1. During verification we ask FuzzBALL for satisfying assignments on execution paths where the behavior of f_2 differs from that of f_1.

More concretely, say that f_1 takes arguments $(x_1, ..., x_n)$ and f_2 takes arguments $(y_1, ..., y_m)$. To compare f_1 and f_2 for equivalence, we construct a program that branches on whether their outputs match on a specified input. Such a program may have the following structure:

```
void compare(x1, ..., xn) {
    r1 = f1(x1, ..., xn);
    r2 = f2(a(x1, ..., xn));
    if (r1 == r2) { printf("Match\n"); }
    else { printf("Mismatch\n"); }
}
```

Using FuzzBALL, we can determine what values of a and $x_1, ..., x_n$ cause each direction of the branch to be taken. During synthesis we are interested in the 'match' branch and during verification we are interested in the 'mismatch' branch. A candidate adaptor a is generated by executing the above program with a symbolic and $x_1, ..., x_n$ concrete (the values of $x_1, ..., x_n$ come from previously generated counterexamples). When FuzzBALL explores the 'match' branch under these conditions, a valid assignment to a will be an adaptor that makes the outputs of f_1 and f_2 match on all generated test cases. Counterexamples are generated by executing the above program with $x_1, ..., x_n$ symbolic and a concrete (here a is a previous candidate adaptor). When FuzzBALL explores the 'mismatch' branch under these conditions, a valid assignment to the symbolic variables $x_1, ..., x_n$ will be an input on which the outputs of f_1 and f_2 are not equal with the provided adaptor. We indicate to FuzzBALL that $x_1, ..., x_n$ are symbolic by tagging the registers or stack locations where those variables are stored as symbolic memory. How exactly a is represented and made symbolic will depend on the type of adaptor used.

Chapter 4: Results

We implemented the technique described in the previous chapter to compare functions at the binary level on the Linux x86-64 platform. This chapter presents one type of adaptor that our implementation supports: a family of adaptors allowing arithmetic relations between integer values. This type of adaptor allows us to compare functions that take integer arguments.

We begin this chapter with a discussion of the family of 'integer arithmetic adaptors' and conclude with comments on our implementation's performance when synthesizing adaptors of this type.

4.1 Integer Arithmetic Adaptors

Integer arithmetic expressions are a popular target for traditional synthesis because they have a simple form, but often require significant programmer effort. For example, many bit manipulation and array index expressions can be expressed as integer arithmetic expressions. We can use integer arithmetic expressions to construct *integer arithmetic adaptors* between functions that take integer inputs.

We represent integer arithmetic expressions using binary trees whose nodes are constant values, variables, or operators. The operators we allow are listed in Table 4.1. We compute the value of an expression tree in the natural way:

- If the root of the tree is a constant or variable, then the value of that tree is that constant or variable.
- If the root of the tree is a binary operator, then the value of that tree is that operator applied to the values of the left and right subtrees.

- If the root of the tree is a unary operator, then the value of that tree is that operator applied to the value of the left subtree.

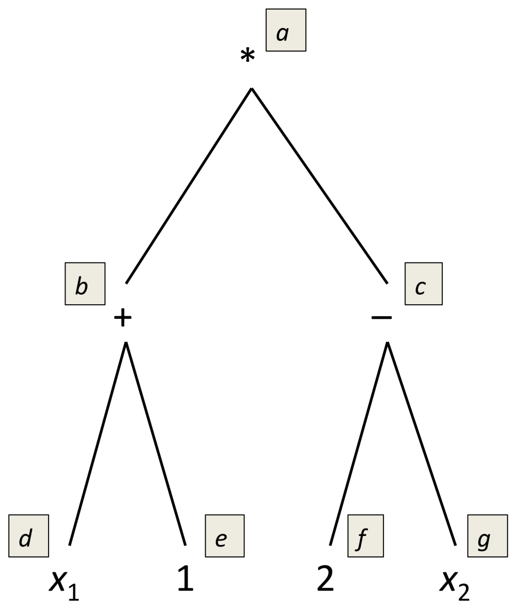
addition ($+$)	subtraction (-)
multiplication (*)	bitwise and (&)
bitwise or ($\|$)	bitwise exclusive or ($\oplus$)
unsigned division ($/_u$)	signed division ($/_s$)
unsigned mod ($\%_u$)	signed mod ($\%_s$)
left shift ($\ll$)	logical right shift ($\gg_u$)
arithmetic right shift ($\gg_s$)	negation (-)
bitwise negation ($\neg$)	

Table 4.1: Integer arithmetic operators supported by FuzzBALL. Note that the division, mod, and shift operators require special handling to avoid division or mod by zero and shifts by an inappropriate value.

To represent an integer arithmetic adaptor between $f_1(x_1, ..., x_n)$ and $f_2(y_1, ..., y_m)$, we use m expression trees whose variables can be any of $x_1, ..., x_n$. To apply an integer arithmetic adaptor to the arguments of f_2, we replace the i^{th} argument with the value of the i^{th} expression tree. As an example, consider $f_1(x_1, x_2) = (x_1 + 1) * (2 - x_2)$ and $f_2(y_1) = y_1$. We represent an integer arithmetic adaptor between f_1 and f_2 as a single expression tree whose variables can be any of x_1, x_2. An expression tree representing a correct adaptor between f_1 and f_2 is given in Figure 4.1. When f_2 is called with the value of the expression tree in Figure 4.1 as input, its behavior will be equivalent to the behavior of f_1.

From FuzzBALL's perspective, an expression tree is a collection of symbolic variables and rules relating those symbolic variables. The symbolic variables indicate the types and values of nodes in the expression tree and the 'rules' are constraints passed to the SMT solver that encode how the expression tree is evaluated. The expression tree in Figure 4.1 is represented by 14 symbolic variables, two for each of the seven nodes.

As the depth of an expression tree increases, the number of symbolic variables needed to represent it increases exponentially. Additional symbolic variables and constraints make it more difficult for the SMT solver to check whether an expression is satisfiable and to find satisfying solutions, so increasing tree depth significantly increases the time required



Node	Type	Value
a	operator	*
b	operator	+
c	operator	-
d	variable	x_1
e	constant	1
f	constant	2
g	variable	x_2

Figure 4.1: A representation of the adaptor $a(x_1, x_2) = (x_1 + 1) * (2 - x_2)$. Each node is associated with two symbolic variables that correspond to that node's type and value. For example, the root node a has two associated symbolic variables, a_{type} and a_{val}, that indicate that it is an operator that corresponds to multiplication. We consider this expression tree to have depth 3.

to synthesize candidate adaptors. We consider how tree depth impacts performance in Section 4.2.

4.1.1 Examples

In this section we provide examples of functions that can be made equivalent using integer arithmetic adaptors.

4.1.1.1 Rectangle Example

Consider two functions, f_1 and f_2, that check whether a point (x, y) is inside of a rectangle. We may think of f_1 and f_2 as functions used by a GUI application that needs to check whether the cursor is in a certain region of the display. Let f_1 and f_2 be defined by:

```
int f1(int x, int y) {
    return (0 <= x) & (x <= 1) & (0 <= y) & (y <= 1);
}
int f2(int x, int y) {
        return (2 <= x) & (x <= 4) & (1 <= y) & (y <= 4);
}
```

The rectangles corresponding to f_1 and f_2 are drawn in Figure 4.2. Our goal is to find an adaptor over the arguments of f_2 that makes the behavior of f_2 equivalent to that of f_1.

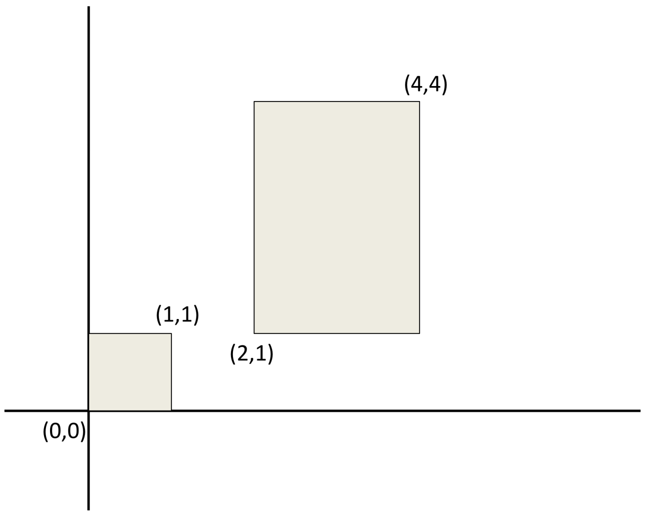


Figure 4.2: The rectangles of interest for f_1 and f_2. We want to determine whether the problem of checking whether a point (x, y) is in the square to the left is equivalent to the problem of checking whether an adapted version of (x, y) is in the rectangle to the right.

x		$2x+2$	y		$2y+2$
0	$\rightarrow$	2	0	$\rightarrow$	2
1	$\rightarrow$	4	1	$\rightarrow$	4
2^{31}	$\rightarrow$	2	2^{31}	$\rightarrow$	2
2^{31}+1	$\rightarrow$	4	2^{31}+1	$\rightarrow$	4

Table 4.2: All values of x and y that are mapped into the region $2 \leq x \leq 4$, $1 \leq y \leq 4$ by a_1. a_1 works as intended on inputs (0, 0), (0, 1), (1, 0), and (1, 1), but produces differing behavior between f_1 and f_2 on inputs like (0, 2^{31}), (1, $2^{31}+1$), (2^{31}, 2^{31}), etc. because of integer overflow.

x		$2x+2$	y		$3y+1$
0	$\rightarrow$	2	0	$\rightarrow$	1
1	$\rightarrow$	4	1	$\rightarrow$	4
2^{31}	$\rightarrow$	2	$\frac{2^{32}+2}{3}$	$\rightarrow$	3
2^{31}+1	$\rightarrow$	4			

Table 4.3: All values of x and y that are mapped into the region $2 \leq x \leq 4$, $1 \leq y \leq 4$ by a_2. a_2 works as intended on inputs (0, 0), (0, 1), (1, 0), and (1, 1), but produces differing behavior between f_1 and f_2 on inputs like (0, $\frac{2^{32}+2}{3}$), (2^{31}, 0), etc. because of integer overflow. Note that a_1 and a_2 have different sets of error values

Two plausible adaptors are:

$$a_1(x, y) = (2x + 2, 3y + 1) \tag{4.1}$$

and

$$a_2(x, y) = (2x + 2, 2y + 2) \tag{4.2}$$

However, these adaptors are only correct over mathematical integers. When using fixed-size machine integers, neither of these adaptors will cause f_1 and f_2 to produce equivalent outputs on all inputs. For example, using 32-bit arithmetic $f_1(x, y) = 0$ while $f_2(a_1(x, y)) = 1$ on the input $(x, y) =$ (0, 0x80000000). Similarly, $f_1(x, y) = 0$ while $f_2(a_2(x, y)) = 1$ on the input $(x, y) =$ (1, 0x55555556). See Tables 4.2 and 4.3 for a list of all values of x and y that are mapped onto the rectangle with corners at (2,1) and (4,4) by a_1 and a_2.

Generally, however, we want to think of a_1 and a_2 as being correct adaptors that show that f_1 and f_2 are equivalent. This means that we need to ignore the 'bad' inputs that cause integer overflow. To do this, we restrict the counterexamples that our verifier can produce. Returning to our GUI application, we may require that all counterexamples (x, y) satisfy $0 \leq x \leq 1920$ and $0 \leq y \leq 1080$, because 1920×1080 pixels is a standard screen resolution. With this restriction, points like (0, 0x80000000) will not be considered counterexamples, and f_1 and f_2 can be considered behaviorally equivalent with a_1 and a_2.

Using counterexample restrictions, our tool can synthesize both a_1 and a_2 (using different random seeds) with a depth 3 tree allowing at least the operations for addition and multiplication. Note that in this example, our adaptors between f_1 and f_2 actually introduce new integer overflow bugs that did not exist in the original functions. In general this is not something we want, but the point of this example is to show that our adaptor synthesis technique can allow for equivalence between functions that diverge on some ‘bad’ inputs.

4.1.1.2 General Synthesis

To point out the relationship between adaptor synthesis and traditional program synthesis, we show how to use adaptor synthesis to construct a bit manipulation expression from Hacker’s Delight [35], a popular source of benchmarks for synthesis performance.

To begin, consider the following two functions:

```
int f1(int x) {
    int i;
    if (x == 0) { return x; }
    for (i = 0; i < 8 * sizeof(x); i++) {
        if (x & (1 << i)) { break; }
    }
    return x ^= (1 << i);
}

int f2(int x) {
    return x & (x - 1);
}
```

Both of these functions turn off the rightmost bit in the input x that has the value 1, or return 0 when x is 0. Followed by a zero test, these functions can be used to determine whether an unsigned integer is 0 or a power of 2. The second version of the function is more optimized, but the first is easier to read and more natural to construct. A typical synthesis problem is to generate the second function using the behavior of the first.

We can apply adaptor synthesis to this problem by comparing f_1 as defined above with the identity function f_2 below:

```
int f2(int x) { return x; }
```

Using a depth 3 expression tree, our tool finds that f_1 and f_2 are equivalent with the adaptor $a(x) = x \ \& \ (x - 1)$, which is exactly the expression we wanted to synthesize.

4.1.1.3 C Library Functions

As part of a large-scale experiment, we searched for behaviorally equivalent functions in the GNU C library (glibc) [1]. Although behaviorally equivalent functions in glibc likely have similar implementations, meaning that they are not useful for N-version or N-variant applications, it is still interesting to see that behaviorally equivalent functions exist 'in the wild,' even in widely used and heavily tested code bases. Here we present two pairs of glibc functions that we found to be equivalent under the family of integer arithmetic adaptors.

isupper and islower — *isupper* and *islower* check whether a character is uppercase or lowercase respectively. When the inputs to both functions are representable as unsigned characters (0–255) or EOF, *isupper*(x) is equivalent to *islower*$(x + 32)$ and *islower*(x) is equivalent to *isupper*$(x - 32)$. The behavior of *isupper* and *islower* on inputs that are not EOF and not in the range 0–255 is undefined. This means that we must restrict the counterexamples that the verifier can generate in order to synthesize an adaptor that makes these functions equivalent. Another interesting feature of *isupper* and *islower* is that they return different nonzero values to indicate 'true,' so checking for equivalence between their outputs is not the same as comparing their return values for equality.

killpg and kill — *killpg*, which sends a signal to a process group, and *kill*, which sends a signal to a process or a group of processes, are equivalent with the adaptor $a(x, y) = (-x, y)$ when $x > 0$. The reason for equivalence between these functions is that *killpg* is implemented in terms of *kill*:

```
int killpg (__pid_t pgrp, int sig)
{
    if (pgrp < 0) {
        __set_errno (EINVAL);
        return -1;
    }
    return __kill (-pgrp, sig);
}
```

4.2 Performance

In this section we discuss the performance of our implementation using the example from Section 4.1.1.2 (copied below for convenience). We use this example because the equivalence of the two functions does not depend on any restrictions on counterexamples, and the specification function (f_1) involves branching, which causes our adaptor synthesis tool to take a different number of iterations when run with different random seeds. All timings presented in this section were taken on a machine with 192 GB RAM and Dual Xeon E5-2623 processors running Ubuntu 14.04. However, this machine's resources were not fully utilized in our tests because the intention of this section is not to report measurements of an optimized implementation, but rather to highlight general techniques for improving performance.

```
int f1(int x) {
    int i;
    if (x == 0) { return x; }
    for (i = 0; i < 8 * sizeof(x); i++) {
        if (x & (1 << i)) { break; }
    }
    return x ^= (1 << i);
}


int f2(int x) { return x; }
```

We know from Section 4.1.1.2 that a correct adaptor between f_1 and f_2 above is $a(x) = x\&(x-1)$. Over 10 trials, with a depth 3 tree allowing the operators +, - (subtraction), &, |, $\oplus$, $\ll$, $\gg_u$, $\gg_s$, - (negation), and $\neg$, synthesizing this adaptor required an average of 17.5 iterations of the verifier/synthesizer loop and 179.7 seconds to complete. The average time required to produce a counterexample for a candidate adaptor was 2.0 seconds, and the average time required to verify that a candidate adaptor was correct was 9.6 seconds. Note the difference in time required for counterexample generation (when the verifier successfully produces a counterexample) and adaptor verification (when the verifier fails to produce a counterexample, indicating that the candidate adaptor is correct). The intuition for this observed difference is it is 'harder' to show that no counterexample exists than it is to produce a counterexample. The average time required to synthesize a candidate adaptor was 8.2 seconds. Candidate adaptor synthesis and counterexample generation times over all 10 trials are graphed in Figure 4.3.

Figure 4.3 shows that while time required to produce a counterexample for an adaptor remained fairly constant over all iterations, the time required to produce a candidate adaptor increased significantly with each iteration. The reason for this is that at each iteration finding a candidate adaptor becomes more difficult because the number of inputs on which the candidate adaptor must have a certain behavior has increased. Figure 4.3 also show that most of the time required for adaptor synthesis is spent on synthesizing candidate adaptors. These observations suggest that to improve adaptor synthesis performance, we want to limit

(1) the total number of iterations required to find a correct adaptor and (2) the time required to synthesize a candidate adaptor at any iteration.

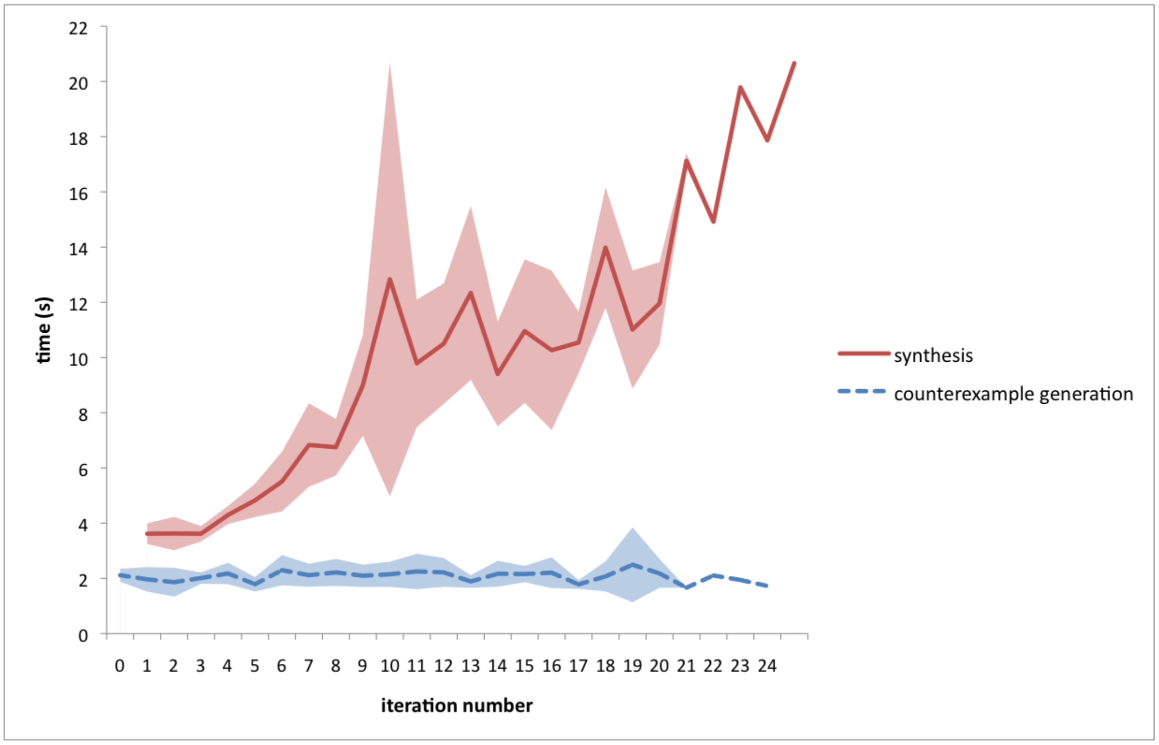


Figure 4.3: Candidate adaptor synthesis and counterexample generation times when synthesizing the expression $x\&(x-1)$. The lines are average times over 10 trials and the shaded regions represent variability in our measurements using standard deviation. There is less variability in the measurements of later iterations because few trials required more than 20 iterations to terminate. The average adaptor verification time (not shown here) was 9.6 seconds.

4.2.1 Restrictions on Expression Trees

4.2.1.1 Reducing the Number of Iterations

One way to limit the number of iterations required to find a correct adaptor is to restrict the types of adaptors that our tool can generate. We restrict integer arithmetic adaptors by limiting the operators and constant values that can appear within their expression trees. The effects of limiting operators and constant values in our example are reported in Table 4.4.

number of operators	number of iterations	total time	counterexample generation time	verification time	candidate adaptor synthesis time
2	11.4	65.0s	1.9s	9.2s	3.4s
5	14.6	91.0s	2.0s	9.4s	4.0s
7	19.2	127.0s	1.9s	8.6s	4.6s
10	17.5	179.7s	2.1s	9.6s	8.2s

size of constants	number of iterations	total time	counterexample generation time	verification time	candidate adaptor synthesis time
2 bit	4.6	30.6s	2.0s	10.8s	3.5s
8 bits	11.4	93.2s	2.0s	10.5s	5.9s
16 bits	12.4	102.8s	1.9s	9.6s	6.3s
24 bits	15.2	136.4s	1.9s	9.3s	7.0s
32 bits	17.5	179.7s	2.1s	9.6s	8.2s

Table 4.4: Performance over 5 trials when synthesizing the expression $x\&(x-1)$ with different restrictions on the operators and constant values allowed in the arithmetic expression tree. In the top table all constant values are allowed, but the operators are restricted to be the first i operators in the list [- (subtraction), &, +, |, $\oplus$, - (negation), $\neg$, $\ll$, $\gg_u$, $\gg_s$]. In the bottom table all 10 original operators are allowed, but constant values are restricted to be in the range $(-2^{i-1}, 2^{i-1}-1)$.

The main takeaway from Table 4.4 is that the number of iterations decreased as we restricted operators and constant values, which led to lower total running time. Note that when restricting the operators allowed in an expression tree, running time depends not only on the number of operators allowed, but also the types of operators allowed. For example, operations like multiplication, division, and mod are inherently more expensive for SMT solvers to reason about. Shifts by arbitrary values are also expensive, which is why in Table 4.4 the total adaptor synthesis time increased when we switched from using 7 operators to 10 operators (by including shifts) even though the number of iterations decreased. The apparent decrease in candidate adaptor synthesis time in Table 4.4 as we restrict operators and constant values is a result of the decrease in the number of iterations.

Restricting adaptor inputs (which is the technique we have used so far to exclude certain counterexamples) can also decrease running time by reducing the number of iterations. However, the adaptors generated in this case are only guaranteed to work on the restricted input domain. For example, when we restricted the adaptor inputs in our example above to be in the range 0–100, the average number of iterations required over 5 trials was 9.6 (as opposed to 17.5), the average total running time was 62.4 seconds (as opposed to 179.7 seconds), and the following adaptors were synthesized:

- $a(x) = x$ & (x - 0xc94e8f81)
- $a(x) =$ (x - 0x802a8081) & x
- $a(x) =$ (x - 0x7c898101) & x
- $a(x) = (x \oplus$ 0xe2322e00) & (0xff + x)
- $a(x) = (x +$ 0xa6ffffff) & (x - 0xf0000000)

It turns out that none of these adaptors are correct over all 32-bit integers. For example, none of them produce the correct result on the inputs 0xffffffff or 0x1000100. In fact, they all produce incorrect results on many of the inputs outside the range 0–100. This indicates that adaptor inputs should only be restricted as a means of ignoring inputs on which we allow functions to have different behaviors, and should not be restricted for the purpose of improving synthesis performance.

4.2.1.2 Reducing Candidate Adaptor Synthesis Time

One way to reduce the time required to synthesize candidate integer arithmetic adaptors is to decrease the expression tree depth. We cannot decrease the tree depth in our example and still synthesize an adaptor that makes f_1 and f_2 equivalent (because there is no integer arithmetic adaptor of depth 2 between f_1 and f_2 that makes them equivalent), so we instead consider the effect of re-running our original experiment with the expression tree depth increased to 4. Candidate adaptor synthesis times for the depth 3 and depth 4 trees are plotted in Figure 4.4.

We see in Figure 4.4 that at iteration 20 the time required to synthesize a candidate adaptor with a depth 4 tree was almost 100 times the time required to synthesize a candidate

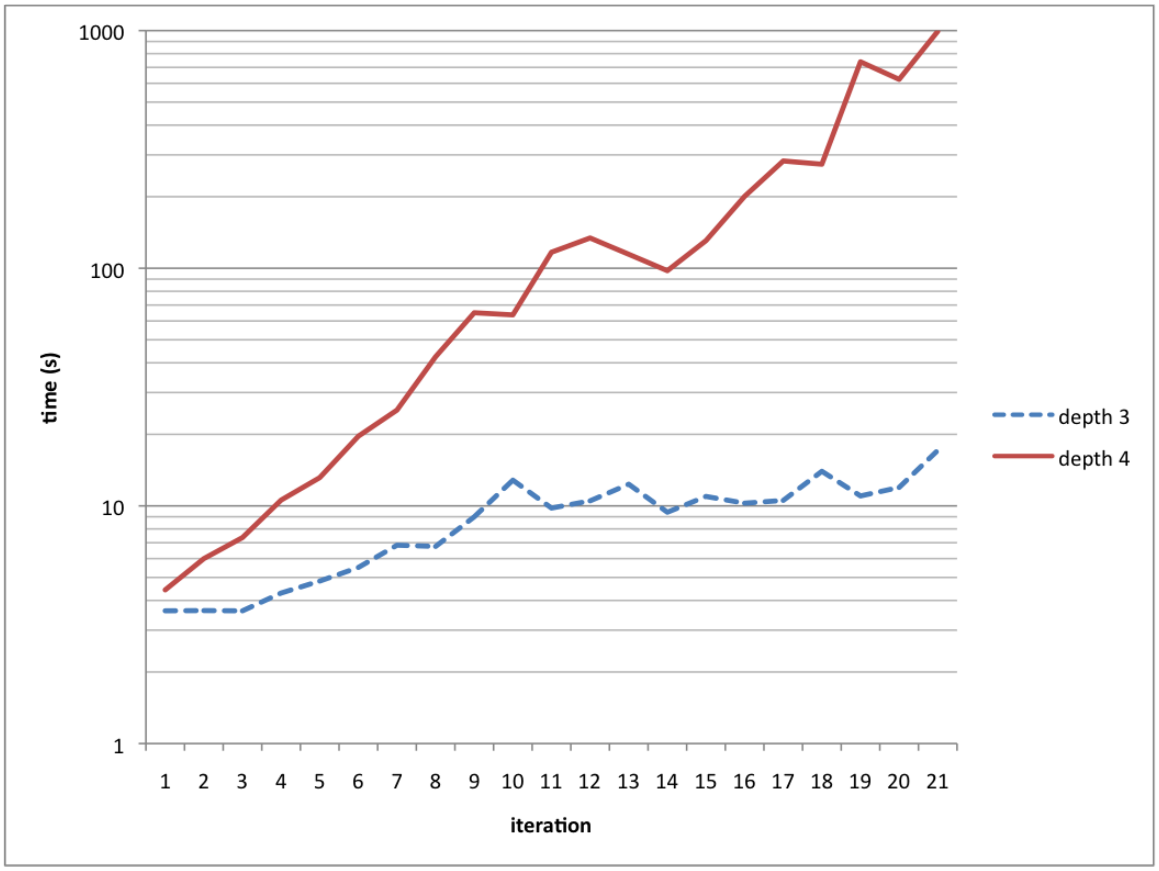


Figure 4.4: A comparison of the time required to synthesize candidate adaptors at different iterations using a depth 3 and depth 4 tree. Note that time is shown on a log scale.

adaptor with a depth 3 tree allowing the same operators and constant values. The reason for this substantial difference in performance is that increasing tree depth creates exponentially more variables that the SMT solver has to reason about. This indicates that for synthesis of integer arithmetic adaptors, we always want to use the smallest tree depth possible.

4.2.2 Caching SMT queries

At each iteration, the synthesizer component of our adaptor synthesis tool tries to find an adaptor that makes two functions produce equivalent output on every input in the list of current counterexamples. The list of current counterexamples grows by one with every iteration, but aside from this new element the list of counterexamples remains the same.

This means that the synthesizer must build many of the same symbolic formulas and make many of the same queries to the SMT solver as it did in the previous iteration.

A simple way to avoid redoing this work is to store previously seen responses to SMT queries in a cache. That way, when the synthesizer makes a query that it made before, there is only a short delay in reponse time. This simple method is effective because the majority of time spent on candidate adaptor synthesis is spent waiting for replies to SMT queries. In Figure 4.5 we plot the candidate adaptor synthesis and counterexample generation times presented in Figure 4.3 along with with the corresponding adaptor synthesis times without SMT query caching (note that all of the measurements presented so far have used SMT query caching). We can see that without caching, candidate adaptor synthesis takes much longer at later iterations.

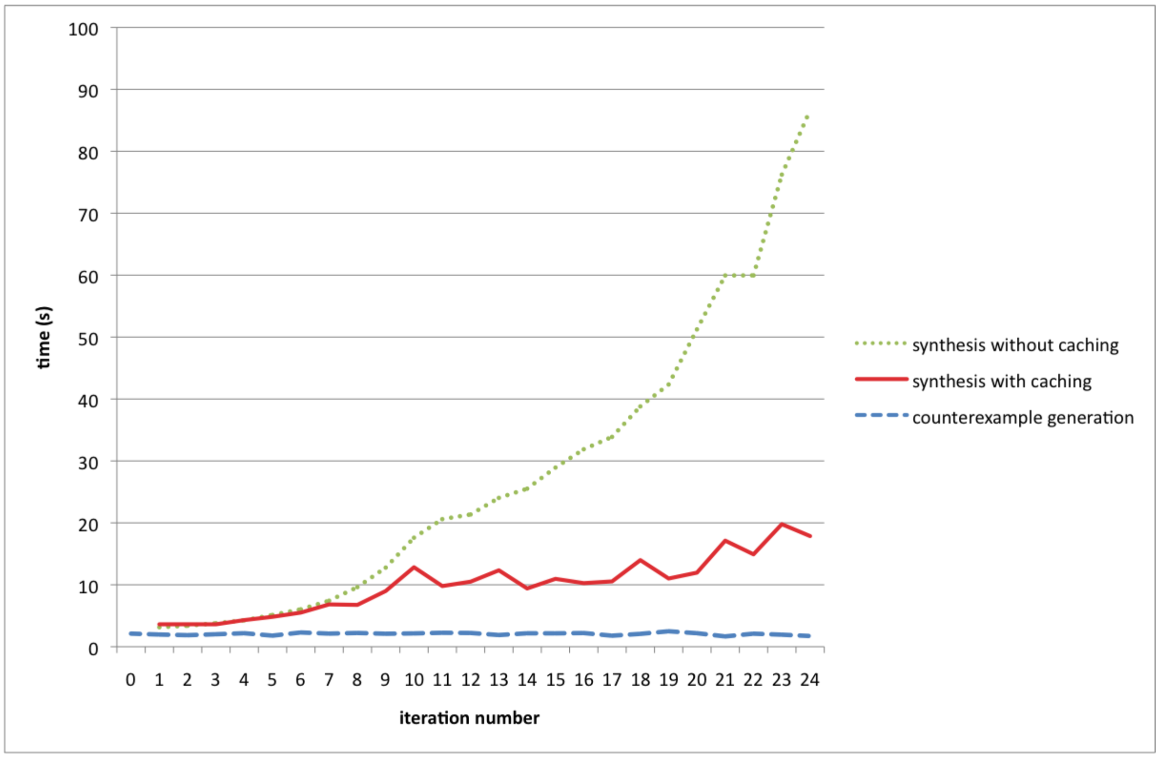


Figure 4.5: A comparison of synthesis performance with and without SMT query caching.

4.2.3 Iterative Search

Our discussion so far has made clear the need for reducing the adaptor search space by restricting arithmetic expression tree depth and allowed operators and constants. However, in practice it is difficult to know what tree depth, operators, and constants will allow for an adaptor between two arbitrary functions that makes those functions equivalent. On one hand we want to support expressive arithmetic adaptors that are likely to allow for equivalence between many functions, but on the other hand we want the adaptor synthesis process to terminate in a reasonable amount of time.

One way to approach this problem is to use an iterative search process, starting with arithmetic adaptors that use small trees allowing a restricted set of operators and constants, and working up towards more complex adaptors as needed.

Chapter 5: Discussion

So far we have presented a general technique for comparing arbitrary functions for equivalence and discussed the application of this technique to functions that take only integer arguments. In this chapter we provide a qualitative comparison of our work with existing tools as well as a discussion of directions for future work.

5.1 Comparison with Existing Tools

Here we compare our approach to that of EQMINER [21], UC-KLEE [30], and SKETCH [33]. EQMINER and UC-KLEE are tools for detecting semantic clones. As discussed in Chapter 2, EQMINER uses random testing to partition code into behaviorally equivalent clusters and UC-KLEE uses symbolic execution to cross-check arbitrary C functions. SKETCH is a program sketching tool that uses a CEGIS style of synthesis.

5.1.1 EQMINER

The fundamental difference between our tool and EQMINER is that EQMINER uses random testing to detect behaviorally equivalent code fragments while we use symbolic execution to detect behaviorally equivalent functions. Random testing is highly scalable and naturally allows for equivalence between diverse implementations of the same functionality, even when those implementations have different error-handling and boundary cases. By contrast, symbolic execution is expensive and may require user-involvement to allow for equivalence between functions that have different behaviors on certain inputs. The primary drawback of using random testing to compare code for equivalence is that while testing can guarantee behavioral non-equivalence on certain inputs, it can rarely guarantee behavioral equivalence on all inputs. This means that the behaviorally equivalent clusters that EQMINER constructs may not actually consist of code fragments that are all behaviorally equivalent. Instead, the clusters group functions that that have equivalent behaviors on a set of selected random inputs (although if two pieces of code have equivalent behaviors on random inputs, it is likely that they have equivalent behaviors on many inputs).

Symbolic execution allows for stronger guarantees about the equivalence of functions. A complete implementation of our adaptor synthesis technique should be able to compare two functions f_1 and f_2 using a family of (finite) adaptors to either (1) produce an adaptor that makes the input/output behavior of f_1 and f_2 equivalent for all execution paths explored or (2) prove that there is no adaptor in the specified adaptor family that makes f_1 and f_2 equivalent. Note that although symbolic execution cannot make guarantees about behaviors on paths not explored (which is related to the inability of testing to guarantee behavior on inputs not tested), it can characterize code behavior for many more inputs than testing.

The other difference between our tool and EQMINER is that we compare functions for equivalence while EQMINER compares code fragments. In order to compare the input/output behavior of code fragments, EQMINER defines inputs as variables that are used but not defined, and outputs as variables that are defined but not used. This definition treats inputs and outputs as unordered sets, which means that EQMINER must allow for equivalence between code fragments that take the same collection of inputs, regardless of the order of those inputs. To this end, EQMINER defines code fragments C_1 and C_2 to be behaviorally equivalent if there exist permutations p_1 and p_2 such that $C_1(p_1(I)) = C_2(p_2(I))$ for all inputs I. The combination of the permutations p_1 and p_2 can be viewed as an adaptor between C_1 and C_2.

5.1.2 UC-KLEE

UC-KLEE is built on top of the KLEE [6] symbolic execution system. Given two functions, UC-KLEE either finds an input on which the behavior of those functions is different, or proves that no such input exists for a finite number of explored paths. In this way, UC-KLEE is similar to the verifier component of our adaptor synthesis tool (in this case, the candidate adaptor being 'verified' is the identity adaptor). UC-KLEE does not allow for equivalence between functions that have different argument structures.

One application of UC-KLEE is to cross-checking implementations of the same functionality that have the same interface. One interesting feature of using UC-KLEE to cross check implementations is that users are not required to write test cases or specifications that describe the functionality that the implementations should provide. Instead, they may need to describe 'don't care' behaviors to UC-KLEE to suppress detection of uninteresting differences between functions. Uninteresting differences include differences in error handling and behavior on invalid inputs. This is a feature of our tool as well— adaptor synthesis does not require any knowledge of the expected behavior of two functions to compare them for equivalence, but it may need to know how restrict the domain of counterexamples so that the behavior of the functions on 'bad' inputs is ignored. Restricting the domain of counterexamples is what allows us to consider diverse function implementations equivalent.

5.1.3 SKETCH

The idea of program sketching is to have the programmer write an incomplete partial program (a 'sketch'), and use synthesis to fill in the missing implementation details. Program sketching is not the intended application of adaptor synthesis, but it is interesting to see that our adaptor synthesis technique can be used for this purpose. To compare our tool to SKETCH, we borrow two examples presented for the SKETCH system and rework them into problems of adaptor synthesis. The examples presented require a family of adaptors that can replace arguments by constant integer values or variables (e.g. the family of integer arithmetic adaptors allowing expression trees of depth 1).

In the first example, we synthesize an efficient implementation of a function that constructs a bitmask isolating the rightmost 0-bit in an input integer. The original SKETCH code is:

```
bit[W] isolate0 (bit[W] x) { // W: word size
    bit[W] ret=0;
    for (int i = 0; i < W; i++)
        if (!x[i]) { ret[i] = 1; break }
    return ret;
}
bit[W] isolate0Sketched (bit[W] x) implements isolate0 {
        return ~(x + ??) & (x + ??)
}
```

To make the behavior of *isolate*0*Sketched* match the specification *isolate*0, SKETCH synthesizes constant integer values to replace the question marks, which are referred to as 'holes.' In this case, the first hole should be filled by 0 and the second hole should be filled by 1. To make this example match the adaptor synthesis examples we have considered so far, we let f_1 be the reference implementation *isolate*0 above, and construct f_2 as follows:

```
int f2 (int x, int q1, int q2) {
    return ~(x + q1) & (x + q2)
}
```

Here q_1 and q_2 are the unknown values that we need to replace with constants. Our tool constructs the adaptor $a(x) = (x, 0, 1)$, which indicates that $f_1(x)$ is equivalent to $f_2(x, 0, 1)$ for any x.

The second example we consider involves polynomial division. The goal is to synthesize a polynomial f such that $(x+1)*(x+2)*f = x^4+6x^3+11x^2+6x$. The original SKETCH code is:

```
int spec (int x) {
    return x*x*x*x + 6*x*x*x + 11*x*x + 6*x;
}
int p (int x) implements spec {
        return (x+1) * (x+2) * poly(3,x);
}
int poly (int n, int x) {
        if (n == 0) return ??;
    else return x * poly(n-1, x) + ??;
}
```

The holes that need to be filled in are in the function *poly*, which generates a polynomial in x of degree at most n. To translate this example into our notation we let f_1 be the reference implementation *spec* and define f_2 as follows:

```
int poly (int x, int q1, int q2, int q3, int q4) {
        return x * (x * (x * q4 + q3) + q2) + q1;
}
int f2 (int x, int q1, int q2, int q3, int q4) {
        return (x+1) * (x+2) * poly(x, q1, q2, q3, q4);
}
```

Note that we have expanded the definition of *poly* for $n == 3$. Our tool constructs the adaptor $a(x) = (x, 0, 3, 1, 0)$ between f_1 and f_2, indicating that $f_1(x)$ is equivalent to $f_2(x, 0, 3, 1, 0)$ for all x. Or, equivalently, $(x+1)*(x+2)*poly(x, 0, 3, 1, 0) = (x+1)*(x+2)*x*(x+3)$ is equivalent to $x^4+6x^3+11x^2+6x$.

5.2 Future Work

There are many interesting directions for future work on this project. We summarize a few below:

New families of adaptors — Some of the most interesting work to be done is in finding efficient ways to represent expressive adaptors. The family of adaptors presented in this paper can only be used with functions that take only integer arguments, and may have poor synthesis performance on complex expressions.

Complete comparison of function outputs — As discussed in Chapter 3, despite the examples presented here, comparing the output of two functions for equivalence is rarely as simple as testing their return values for equality. One reason for this is that functions may have semantically-equivalent return values that are not exactly equal. Another complication is that functions should not be considered equivalent if they have different side-effects. Our current tool allows for basic comparisons of system calls and writes to memory, but does not account for equivalence between the same system call with different arguments, different system calls with related arguments, or equivalent writes to different locations in memory. Comparing system calls and writes to memory for semantic equivalence is a nontrivial problem.

Automated counterexample restrictions — In Section 4.1.1.1 we restricted the counterexamples generated during verification based on assumptions about the intended use of our example functions. In practice, assumptions about how a function will be used are reflected in that function's invariants. Invariants can be automatically recovered from code using variable traces produced while executing that code on a collection of inputs [11].

Finding diverse function implementations — Our adaptor synthesis technique can not tell whether the functions it finds to be behaviorally equivalent are computationally diverse, which is a problem for N-version and N-variant applications where we need to be confident that different program implementations have non overlapping bugs. This indicates a need for the ability to characterize 'how different' behaviorally equivalent functions are from each other. One step in this direction might be to compare behaviorally equivalent functions for syntactic similarity, with the intuition that behaviorally equivalent and semantically similar functions likely have related bugs. (However, comparing functions for syntactic similarity requires that the source code of the functions is available, which is not something that our tool assumes.)

Better adaptors — The adaptors that our tool currently generates are not optimized with respect to any particular property; our tool is equally likely to synthesize any adaptor that makes f_1 and f_2 equivalent. For example, when using the family of integer arithmetic adaptors, our tool may generate the adaptor $a(x) = (\text{0xffffffff} \ \& \ x) \gg (x \oplus x)$ when the simpler adaptor $a(x) = x$ would have sufficed. The iterative search approach described in Section 4.2 could address this issue by giving preference simple adaptors that involve efficient operations.

Chapter 6: Conclusion

We have presented a novel technique for detecting behaviorally equivalent functions, even when those functions have different argument structures. The approach works by automatically synthesizing an adaptor between two functions that makes their input/output behavior equivalent, or proving that no such adaptor exists within a specified family of adaptors. We implemented our approach to compare functions at the binary level on the Linux x86-64 platform. One type of adaptor that our implementation supports is the family of integer arithmetic adaptors, which are constructed from arithmetic operations over integer values. Our preliminary assessments of our implementation suggest that adaptor synthesis is a promising approach to mining diverse function implementations that can be used to construct diverse program implementations for a multivariant execution system.

Bibliography

[1] The GNU C Library (glibc), 2016.

[2] Algirdas Avizienis and Liming Chen. On the implementation of N-version programming for software fault tolerance during execution. In *International Computer Software and Applications Conference*, 1977.

[3] Benoit Baudry and Martin Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Survey*, 48(1):16, 2015.

[4] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 158–168. ACM, 2006.

[5] Hayley Borck, Mark Boddy, Ian J De Silva, Steven Harp, Ken Hoyme, Steven Johnston, August Schwerdfeger, and Mary Southern. Frankencode: Creating diverse programs using code clones. In *Proceedings of the 2016 IEEE 23nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 604–608. IEEE, 2016.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[7] Benjamin Cox and David Evans. N-variant systems: A secretless framework for security through diversity. In Angelos D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006.

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*, volume 5902 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2009.

[9] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In Carey Williamson, Aditya Akella, and Nina Taft, editors, *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, pages 475–488. ACM, 2014.

[10] Dave E. Eckhardt and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Trans. Software Eng.*, 11(12):1511–1517, 1985.

[11] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 213–224. ACM, 1999.

[12] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 321–330. ACM, 2008.

[13] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[14] Miguel Garcia, Alysson Neves Bessani, Ilir Gashi, Nuno Ferreira Neves, and Rafael R. Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Softw., Pract. Exper.*, 44(6):735–770, 2014.

[15] Ilir Gashi, Peter T. Popov, and Lorenzo Strigini. Fault diversity among off-the-shelf SQL database servers. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 389–398. IEEE Computer Society, 2004.

[16] Jin Han, Debin Gao, and Robert H. Deng. On the effectiveness of software diversity: A systematic study on real-world vulnerabilities. In Ulrich Flegel and Danilo Bruschi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 6th International Conference, DIMVA 2009, Como, Italy, July 9-10, 2009. Proceedings*, volume 5587 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 2009.

[17] Les Hatton. N-version design versus one good version. *IEEE Softw.*, 14(6):71–76, November 1997.

[18] Matthias Jacob, Mariusz H. Jakubowski, Prasad Naldurg, Chit Wei Saw, and Ramarathnam Venkatesan. *The Superdiversifier: Peephole Individualization for Software Protection*, volume 5312 of *Lecture Notes in Computer Science*, pages 100–120. Springer, 2008.

[19] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM, 2010.

[20] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 96–105. IEEE Computer Society, 2007.

[21] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In Gregg Rothermel and Laura K. Dillon, editors, *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 81–92. ACM, 2009.

[22] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 304–314. ACM, 2002.

[23] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[24] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.*, 12(1):96–109, 1986.

[25] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 276–291. IEEE Computer Society, 2014.

[26] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 289–302. USENIX Association, 2004.

[27] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[28] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: hi-fi tests for lo-fi emulators. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 337–348. ACM, 2012.

[29] Henry Massalin. Superoptimizer - A look at the smallest program. In Randy H. Katz, editor, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto, California, USA, October 5-8, 1987.*, pages 122–126. ACM Press, 1987.

[30] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 669–685. Springer, 2011.

[31] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. Runtime defense against code injection attacks using replicated execution. *IEEE Trans. Dependable Sec. Comput.*, 8(4):588–601, 2011.

[32] Vaibhav Sharma, Kesha Hietala, and Stephen McCamant. Finding semantically-equivalent binary code by synthesizing adaptors. Submitted to the *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 2016.

[33] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[34] Eric Totel, Frédéric Majorczyk, and Ludovic Mé. *COTS Diversity Based Intrusion Detection and Application to Web Servers*, volume 3858 of *Lecture Notes in Computer Science*, pages 43–62. Springer, 2005.

[35] Henry S. Warren. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.