Adapter Synthesis: Synthesizing and Repairing Programs Using Scalable Symbolic Execution

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Vaibhav Bhushan Sharma

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stephen McCamant

January, 2020

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school.

Dedication

To those who held me up over the years

Abstract

Independently developed codebases typically contain many segments of code that perform the same or closely related operations (semantic clones). Finding functionally equivalent segments enables applications like replacing a segment by a more efficient or more secure alternative. Such related segments often have different interfaces, so some glue code (an adapter) is needed to replace one with the other. We present an algorithm that searches for replaceable code segments by attempting to synthesize an adapter between them from some finite family of adapters; it terminates if it finds no possible adapter.

The use of symbolic execution during adapter search implicitly relies on being able to summarize the entire adapter into a single formula. Merging related execution paths is a powerful technique for reducing path explosion in symbolic execution. One approach, introduced and dubbed "veritesting" by Avgerinos et al., works by statically translating a bounded control flow region into a single formula. Java's typed memory structure is very different from a binary, but we present an extension of previous path-merging approaches for symbolic execution of Java.

Bugs in commercial software and third-party components are an undesirable and expensive phenomenon. Such software is usually released to users only in binary form. The lack of source code renders users of such software dependent on their software vendors for repairs of bugs. Such dependence is even more harmful if the bugs introduce new vulnerabilities in the software. Being able to automatically repair security bugs breaks this dependence and increases software robustness. In the third part of this dissertation, we present an automated program repair approach for binary code.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Program synthesis is a technique for automatically writing programs that conform to a given specification [1, 2]. It has the potential to provide an answer to the popular question: can robots replace software developers? However, for solving practical problems, program synthesis often turns into into using deductive reasoning to solve intractable problems, given the large space of potential implementations that have to be considered. Syntax-guided synthesis [3] is a program synthesis technique that attempts to address scalability challenges faced by program synthesis. Given a specification and a syntactic template for the program, syntax-guided synthesis attempts to synthesize a program that fits the syntactic template and satisfies the given specification. In the first part of this dissertation, we present a technique named *Adapter Synthesis*, an extension of syntax-guided synthesis. Given a specification of the desired implementation and a context-free grammar that can be used to synthesize expressions, adapter synthesis synthesizes an *adapter* expression, that wraps around another implementation, to make it satisfy the given specification. The adapter expression is synthesized from the given context-free grammar. We term the implementation that is wrapped around by an adapter the *reference* implementation.

While the target correctness specification can be be specified as a property, adapter synthesis uses an existing implementation as the specification. We term the code that provides an implementation of a correctness specification as the *target* implementation.

This assumption gives adapter synthesis the advantage of being able to run the target code with inputs to generate expected outputs of the specification. When implemented at the binary level, we find that this assumption also enables applications of adapter synthesis such as reverse engineering and deobfuscation. If an adapted reference implementation satisfies the specification described by some arbitrary binary code, and source code is available for the reference implementation, then an adapter synthesis user can simply read the source code to understand the functionality provided by the arbitrary binary code. Since adapter synthesis synthesizes code that wraps around a reference implementation, we find that adapter synthesis helps syntax-guided synthesis scale to large programs, even allowing substitutability of functionality from an entire library. We present adapter synthesis and its applications in Chapter 2.

Adapter synthesis requires a user to specify a family of expressions from which an adapter expression should be synthesized. In order to provide a user of adapter synthesis the flexibility to specify an arbitrary context-free grammar, it is desirable to allow the adapter grammar to be specified at the source-level. However, our implementation of adapter synthesis uses symbolic execution of target and reference code to search for adaptable substitutability. Symbolic execution performs non-standard execution of programs assigning inputs to special variables that simultaneously represent a large set of input values. When symbolically executing adapter grammar, symbolic execution runs into path explosion since it has to explore the possibility of every candidate adapter expression. Path-merging [4, 5] is a known way to alleviate the path explosion problem. Previous work [4] proposed a technique named "veritesting" that statically summarizes multi-path code regions. The summaries constructed by veritesting are as large as possible with summarization often stopped at program locations that cannot be statically summarized. In this dissertation, we propose a reinterpretation and extension of veritesting for symbolic execution of Java bytecode. We provide our implementation in a tool named *Java Ranger* [6, 7].

When a symbolic executor runs into a symbolic branch condition with two feasible sides, Java Ranger attempts to automatically summarize the side-effects of symbolically executing both sides of the branch condition until control-flow from both sides of the branch reaches a common program location. We term the region of code that lies

between the symbolic branch condition and the common program location as a *multi-path region*. Summarization of multi-path regions includes field and array accesses, method calls, and summarizing multiple return values from methods into a single return value. Java Ranger summarizes all of these Java constructs when they appear in a multi-path region along with allowing other unsummarized Java constructs to still appear in a multi-path. In order to summarize multi-path regions containing such constructs, Java Ranger transfers control from the summarization back to the symbolic executor to symbolically execute such unsupported code. We present Java Ranger along with its evaluation and empirical assessment of correctness in Chapter 4

While adapter synthesis is able to scale to larger correctness specifications provided by a target implementations, it needs to have a library of reference implementations around which it synthesizes wrappers. This need for a library of reference implementations is a mixed blessing: it improves the scalability of program synthesis but introduces a dependency on finding other implementations of the target specification. This dependency can be dropped if the goal of adapter synthesis is changed to one of repairing bugs in the target implementation. In this setting, given a buggy implementation of the target specification, adapter synthesis can be reduced to program synthesis that attempts to synthesize changes in the buggy implementation that repair its buggy behavior, while preserving the non-buggy behavior. Using program synthesis for finding the needed repair avoids the need for finding non-buggy reference implementations. It is useful for finding repairs that involve small code changes to the buggy implementation. Finally, applying program synthesis at the binary level allows repair of arbitrary binary code without any need for its source code. This application of program synthesis puts the control of repairing buggy binary-only commercial software in the hands of the software's users instead of relying on the software's vendor to release fixes. We explore this application of program synthesis for repairing insecure binary programs in Chapter 5.

1.2 Background

In this section, we will explain some background knowledge that will be useful in understanding Chapters 2, 4, and 5. We will first explain the Counterexample-Guided

Inductive Synythesis (CEGIS) algorithm. This algorithm is used for both adapter synthesis and program repair in Chapters 2 and 5 respectively. Next, we will explain symbolic execution which is used across all the chapters that follow in this dissertation.

1.2.1 Counterexample-Guided Inductive Synthesis

Program synthesis, by definition, has to involve a component that searches for a implementation in the space of candidate implementations. The implementation that matches the desired specification is presented as the final answer. There are many approaches used in syntax-guided synthesis for searching for an implementation. These involve enumerating possible solutions in order of increasing complexity [8], constraint-based learning [9, 10], using stochastic search [9, 11], and using classifiers with geometry [12]. However, one step which remains in common with all of these aforementioned learning strategies is the need for a teacher that, given a synthesized implementation, certifies it as being correct or returns a counterexample that proves the synthesized implemnetion as being incorrect. Adapter synthesis also makes use of a teacher/learner model which is commonly referred to as the *Counterexample-Guided Inductive Synthesis* in inductive learning. We present the outline of Counterexample-Guided Inductive Synthesis in Figure 1.1.

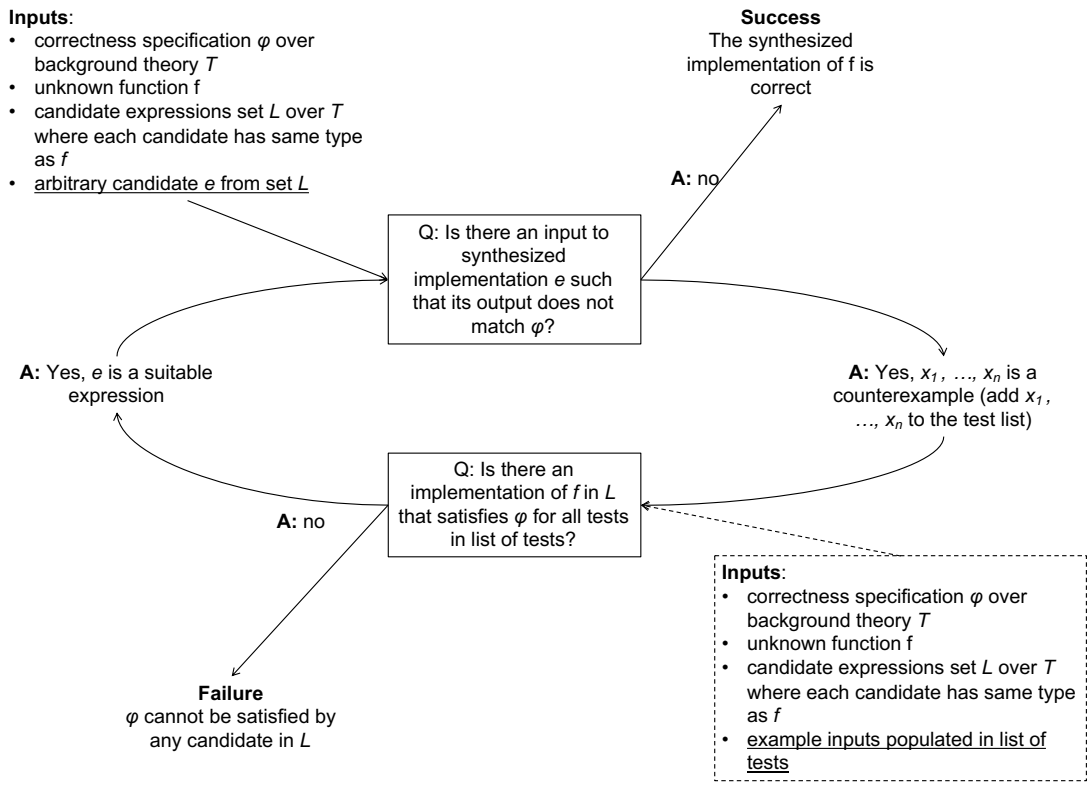


Figure 1.1: Counterexample-guided inductive synthesis

The CEGIS algorithm inductively learns from a set of previously found counterexamples. To begin, it has to be given a correctness specification, φ over a background theory T. The background theory is used to build the set of candidate expressions L. CEGIS also needs to be given the unknown function f which has the same type as its potential candidates.

CEGIS can begin with an arbitrary expression e from the set of candidate expressions L as shown in the top left corner. In this case, CEGIS would first attempt to invalidate the arbitrary expression e by looking for a counterexample. If no such counterexample is found, then the expression e is certified to be correct and produced as the final answer. If a counterexample is found, then CEGIS proceeds by adding it to a list of tests and next checking if it can find a different expression e from the set L of candidate expressions that satisfies the correctness specification φ. If no such expression can be found, CEGIS gives up and concludes that no expression can be synthesized

from the set L of candidate expresssions. But, if a candidate expression e is found, CEGIS returns back to the step of verifying e. Instead of beginning from the top left corner, CEGIS can also begin from the bottom right corner by adding a set of previously known example inputs to a list of test cases. In this case, CEGIS will first attempt to find an expression e in the set L of candidate expressions that satisfies the correctness specification *varphi* for all inputs in the list of tests. CEGIS has been used for both adapter synthesis and program repair in Chatpers 2 and 5 in this dissertation.

1.2.2 Symbolic Execution

CEGIS performs inductive learning that often makes use of two independent deductive procedures to learn an expression and verify it. Symbolic execution [13, 14], coupled with SMT solving, is one such deductive procedure that can be used with one or both steps in CEGIS. Our implementation of adapter synthesis uses symbolic execution as the deductive procedure in CEGIS. Symbolic execution performs nonstandard execution of a program by replacing the inputs to the program with symbolic values. A symbolic value represents a set of zero or more values that the input can have when the program is run without symbolic execution. We present an example in Listing 1.1 along with the tree of possible choices explored by symbolic execution in Figure 1.2.

```
1  // a, b, c are set to symbolic inputs
2  int function (int a, int b, int c) {
3    int output=0;
4    if ( b && c) output += 1;
5    if ( c ) {
6       if ( !b && a)
7          output +=1;
8      output += 2;
9   }
10  if(output == 3) assert(0);
11  return output;
12 }
```

Listing 1.1: C code that sets its three inputs, `a`, `b`, `c` as being symbolic

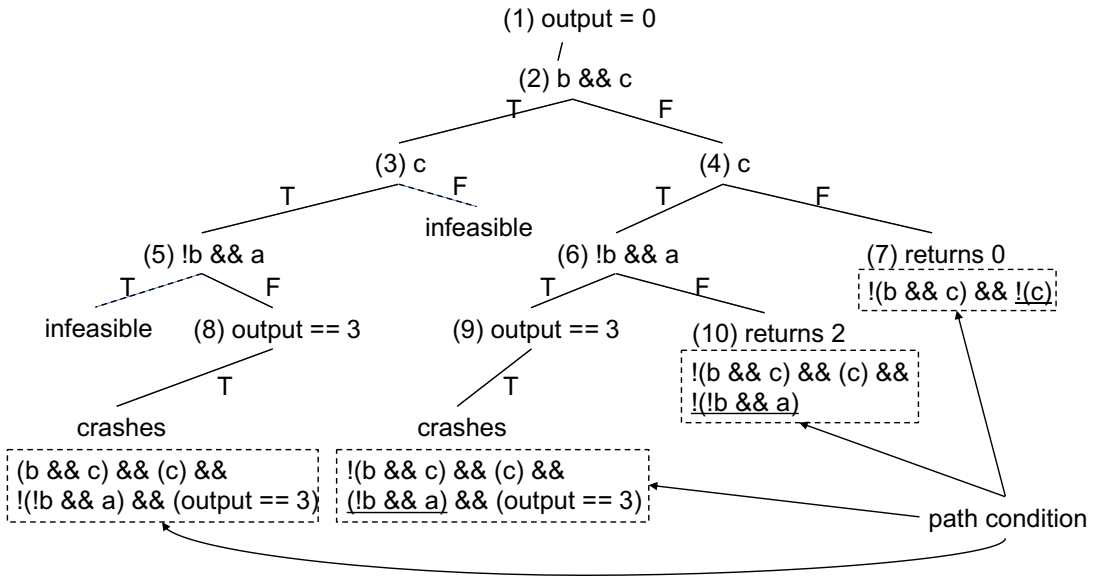


Figure 1.2: Tree of choices explored by symbolic execution when executing the example in Listing 1.1 with the inputs `a`, `b`, `c` set to symbolic values.

Symbolic execution of the code shown in Listing 1.1 begins by setting the three inputs to the function `a`, `b`, and `c` to be symbolic. It then proceeds to executing every line of code, for example, by setting the local variable, `output`, to the concrete value, `0` as shown in Node 1 of Figure 1.2. The branches on lines 4, 5, and 6 of Listing 1.1 are branches which involves conditions with symbolic operands. On encountering a symbolic branch, symbolic execution tools internally use a Satisfiability Modulo Theories (SMT) solver to check if either side of the branch condition are feasible. This causes a symbolic executor to conclude both sides of the branch on line 4 are feasible as represented by Node 2 in Figure 1.2. At this point, a symbolic executor has two choices: (1) fork into two processes, one which explores the true side of this branch and the other which explores the false side, (2) explore one side of the branch first and backtrack later to explore the other side. Regardless of which side is taken, let us proceed assuming the symbolic executor takes the true side of the `b && c` branch. The next branch encountered by a symbolic executor is on line 5 of Listing 1.1 as shown in Node 3 of Figure 1.2. Since, at node 3, the true side of the `b && c` branch has previously been followed by the symbolic executor, the symbolic executor will find only the true side of the `c` branch to

be feasible. This is shown in Node 3 of Figure 1.2. Next, the symbolic executor runs into another symbolic branch on line 6. Once again, because the symbolic executor has previously taken the true side of the `b && c` branch, only the false side of the `!b && a` branch is found to be feasible as shown in Node 5 of Figure 1.2. Finally, the symbolic executor runs into branch on line 10 which crashes the program if the local variable `output` equals `3`. At this point, the symbolic executor has finished exploration of one control-flow path in the program. We refer to this as an *execution path* in the rest of this dissertation. The bottom left corner of Figure 1.2 shows the condition that has to be true for the symbolic executor to have reached the end of this execution path. This condition is referred to as the *path condition*. The symbolic executor maintains the path condition along each explored execution path. After completing exploration along one path, the symbolic executor backtracks and explores other unexplored choices in its exploration tree. On encountering each symbolic branch, it checks if one or both sides of the branch are feasible with respect to the path condition for that execution path. It finally concludes that there is another execution path along which a crash occurs and two other execution paths which return a concrete value as the function's output.

1.3 Overview

The rest of this dissertation presents adapter synthesis and its applications in Chapter 2. This work was published in the proceedings of the International Conference on Software Testing, Validation and Verification 2018 [15]. An extended version of this work was published in the Transactions of Software Engineering, the early access version of which is available as of the writing of this dissertation [16]. Next, we address usability challenges of adapter synthesis by proposing use of a path-merging symbolic executor in Chapter 3. In Chapter 4, we present an extension of path-merging for symbolic execution of Java bytecode. The tool presented in Chapter 4 participated in the International Competition on Software Verification 2020. The manuscript accompanying the participation will be appearing soon [6] and the Java Ranger tool is also publicly available [7]. We explore the use of program synthesis for automatically repairing binary programs in Chapter 5. We conclude the thesis proposed in this dissertation in Chapter 6.

Chapter 2

Finding Substitutable Binary Code By Synthesizing Adapters

2.1 Introduction

Given the large corpus of software available today to an average programmer to reuse, it is desirable to reuse well-tested, bug-free chunks of code that implement some required functionality. Finding such code can be difficult to automate, and there is no guarantee that the code found by such a search will have the exact interface the programmer intends to use. At such times, programmers find themselves writing wrapper code and creating unit tests to check if the wrapped code works as they intended. In this chapter, we present a technique named *Adapter Synthesis* [16, 17] that automates the process of finding functions that match the behavior specified by an existing function, while also synthesizing the necessary wrapper needed to handle interface differences between the original and discovered functions. Use cases for our improved technique include replacing insecure dependencies of off-the-shelf libraries with bug-free variants, reverse engineering binary-level functions by comparing their behavior to known implementations, and locating multiple versions of a function to be run in parallel to provide security through diversity [18].

Our technique works by searching for a wrapper that can be added around one function's interface to make it equivalent to another function. We consider wrappers that transform function arguments and return values. For example, Listing 2.1 shows

implementations of the *isalpha* predicate, which checks if a character is a letter, in two commonly-used libraries. Both implementations follow the ISO C standard specification of the *isalpha* function, but the glibc implementation indicates the input is a letter by returning 1024, while the musl implementation returns 1 in that case. The C standard only mandates that the failures of the predicates is to be indicated with a return value of 0.

```
int musl_isalpha(int c) {
  return ((unsigned)c|32)-'a' < 26;
}
int glibc_isalpha(int c) {
  return table[c] & 1024;
}
int adapted_isalpha(int c) {
  return (glibc_isalpha(c) != 0) ? 1 : 0;
}
```

Listing 2.1: musl and glibc implementations of the *isalpha* predicate and a wrapper around the glibc implementation that makes it equivalent to the musl implementation

The glibc implementation can be adapted to make it equivalent to the musl implementation by replacing its return value, if non-zero, by 1 as shown by the *adapted_isalpha* function. This examples illustrates the driving idea of our approach: to check whether two functions, f_1 and f_2, have different interfaces to the same functionality, we can search for a wrapper that allows f_1 to be substituted by f_2. If such a wrapper exists, then we can conclude that f_1 and f_2 implement the same functionality.

We refer to the function being wrapped around as the *reference* function and the function being emulated as the *target* function. In the example above, the reference function is *glibc_isalpha* and the target function is *musl_isalpha*. We refer to the wrapper code automatically synthesized by our tool as an *adapter*. Our adapter synthesis tool searches in the space of all possible adapters allowed by a specified adapter family for an adapter that makes the behavior of the reference function f_2 equivalent to that of the target function f_1. We represent that such an adapter exists by the notation $f_1 \leftarrow f_2$. Note that this adaptability relationship is not symmetric: $f_1 \leftarrow f_2$ does not imply $f_2 \leftarrow f_1$. In the *isalpha* example, the other direction of adapter could be

implemented by multiplying the musl implementation's return value by 1024, but in general only one direction of adaptation may be possible. To efficiently search for an adapter, we use counterexample guided inductive synthesis (CEGIS) [10]. An adapter family is represented as a formula for transforming values controlled by parameters: each setting of these parameters represents a possible adapter. Each step of CEGIS allows us to conclude that either a counterexample exists for the previously hypothesized adapter, or that an adapter exists for all previously found tests. We use binary symbolic execution both to generate counterexamples and to find new candidate adapters; the symbolic execution engine internally uses a satisfiability modulo theories (SMT) solver. We also implement adapter search using a randomly-ordered enumeration of all possible adapters. We always restrict our search to a specified finite family of adapters.

Our implementation of adapter synthesis [17] can adapt not only arguments and the return value of binary code found in registers but also in memory. Our evaluation can be publicly accessed[1] along with all the changes required to the symbolic executor for adapter synthesis[2] . We demonstrate the use of adapter synthesis for the following applications.

- We demonstrate the use of adapter synthesis for adaptably substituting a buggy function with its bug-free variant by finding adaptable equivalence modulo a bug.
- We demonstrate substitution of RC4 key structure setup and encryption functions in mbedTLS (formerly PolarSSL) and OpenSSL by means of synthesizing adapters between their interfaces.
- We present an intra-library evaluation of adapter synthesis by evaluating two of our adapter families on more than 13,000 pairs of functions from the GNU C library.
- We demonstrate the use of adapter synthesis for reverse engineering at scale using binary symbolic execution-based adapter synthesis. We show that code fragments in a ARM-based 3rd party firmware image for the iPod Nano 2g device

[1] Scripts and instructions to drive each evaluation available here: https://github.com/vaibhavbsharma/fuzzball-synth

[2] Changes to FuzzBALL for adapter synthesis available here: https://github.com/vaibhavbsharma/fuzzball-adaptersynth

can be adaptably substituted by reference functions written for the VLC media player [19]. In this evaluation, we complete more than 1.17 million synthesis tasks, with each synthesis task navigating an adapter search space of more than 1.353 x 10^{127} adapters using an estimation from our evaluation in Section 2.4. Enumerating this search space concretely would take an infeasible amount of time (10^{14} years). We find our adapter synthesis implementation finds several instances of reference functions in the firmware image.

The rest of this chapter is organized as follows. Section 2.2 presents our algorithm for adapter synthesis and describes our adapter families. Section 2.3 describes our implementation, and Section 2.4 presents examples of application of adapter synthesis, large-scale evaluations, and a comparison of two adapter search implementations. Section 2.5 discusses limitations and future work, Section 2.6 describes related work, and Section 2.7 concludes.

2.2 Adapter Synthesis

2.2.1 Algorithm for Adapter Synthesis

The idea behind CEGIS is to alternate between synthesizing candidate expressions and checking whether those expressions meet a desired specification. When a candidate expression fails to meet the specification, a counterexample is produced to guide the next synthesis attempt. In our case, the expressions to be synthesized are adapters that map inputs of the target function to inputs of the reference function and outputs of the reference function to outputs of the target function in such a way that the behavior of the two matches. Our specification for synthesis is provided by the behavior of the target function and we define counterexamples to be inputs on which the behavior of the reference function and target function differ for a given adapter.

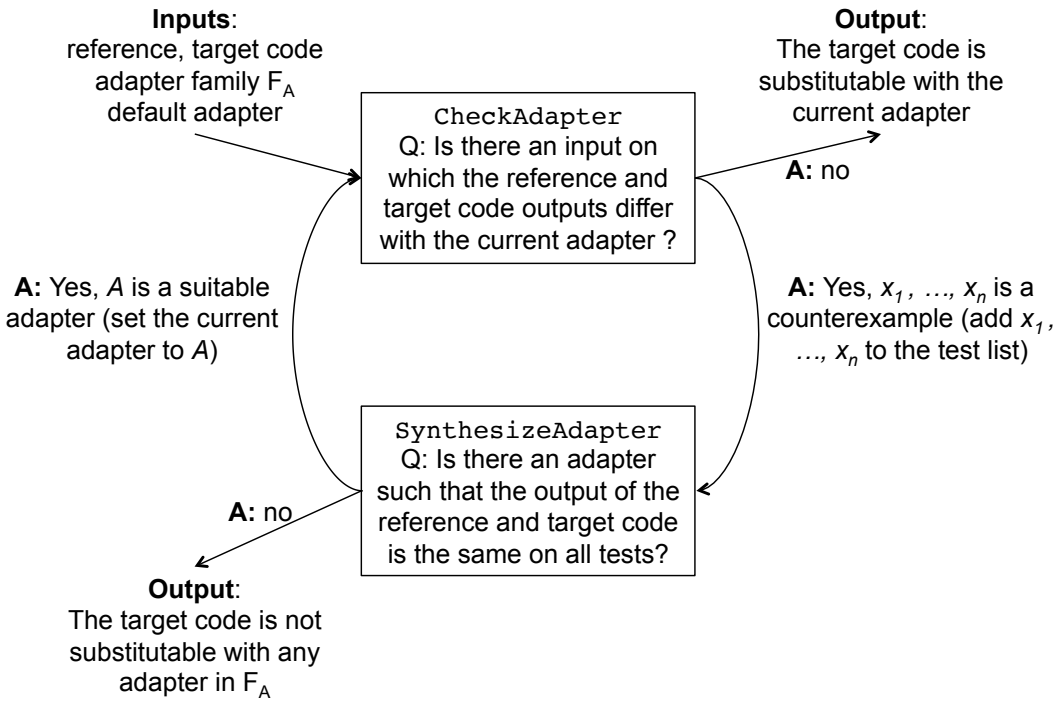


Figure 2.1: Counterexample guided adapter synthesis

Our adapter synthesis algorithm is summarized in Figure 2.1. As shown in Figure 2.1, our algorithm takes as input the reference function, the target function, and an adapter family F_A. The algorithm begins with a default adapter and an empty test list. In our implementation, as a default adapter we often use the "zero adapter," which sets every input and output of the reference function to the constant value 0. Until a correct adapter is found or no new adapter can be synthesized, a new counterexample is added to the test list, and any subsequently generated candidate adapters must satisfy all tests in the list. The output of adapter synthesis is either input and output adapters that allow the target function to be substituted by the adapted reference function or an indication that achieving substitutability is not possible using the specified adapter family F_A. Our algorithm is guaranteed to terminate if the space of adapters allowed by F_A is finite [10]. Although the adapter space and input space may be quite large, in practice we observed that, often, when an adapter was found, the number of steps required to find an adapter was small. For more information, see Section 2.4.7.

To generate counterexamples and candidate adapters, our adapter synthesis algorithm uses procedures `CheckAdapter` and `SynthesizeAdapter`, which both rely on symbolic execution. `CheckAdapter` uses symbolic execution to find an input value on which the target function and reference function outputs disagree with a given adapter and `SynthesizeAdapter` uses symbolic execution to search for adapters that cause the target function and reference function to have equivalent output on all inputs in the current test list. Pseudocode for `CheckAdapter` and `SynthesizeAdapter` can be found in Section A.0.1 in the Appendix.

Adapter synthesis terminates when either `CheckAdapter` or `SynthesizeAdapter` have explored every available execution path without finding a counterexample or new candidate adapter. If `CheckAdapter` fails to find a counterexample, we conclude that the current adapter allows the target function to be substituted by the reference function. If `SynthesizeAdapter` fails to generate an adapter, we conclude that the target function is not substitutable by the reference function with any adapter in F_A. If either `CheckAdapter` or `SynthesizeAdapter` fail due to a timeout, we make no claims about the substitutability of the target function by the adapted reference function. However, our observation from our evaluation is that the majority of adapter synthesis tasks that timed out would eventually lead to the conclusion of no possible adapter. We explore timeouts in more detail in Section 2.4.7.

2.2.2 Adapter Families

The `SynthesizeAdapter` step synthesizes an adapter from a finite family of adapters. We currently support the following families of adapters, of which the arithmetic and memory substitution adapter families are newly introduced in this work.

Argument Substitution

This adapter family allows replacement of any reference function argument by one of the target function arguments or a constant. Our adapter synthesis tool implements argument substitution by constructing a formula for every argument. The formula uses a ternary operator which evaluates to a constant or one of the target arguments based on the concretization of the adapter. We present an example of the argument substitution adapter formula in Listing 2.8 in Section 2.3.2. Listing 2.2 presents an adapter that can

be synthesized using simple argument substitution. While this pair of function is not derived from real-world software, this is an interesting example because the functions f_1 and f_2 have a non-intuitive relationship, and it is not immediately obvious that they are equivalent. This family is useful, for instance, when synthesizing adapters between the cluster of functions in the C library that wrap around the *wait* system call as shown in Section 2.4.5.

```
int f1(int x, unsigned y) {
  return (x << 1) + (y % 2);
}

int f2(int a, int b, int c, int d) {
  return c + d + (a & b);
}

int adapted_simple_f2(int x, unsigned y) {
  return f2(y, 1, x, x);
}
```

Listing 2.2: Simple argument substitution adapter example

Argument Substitution with String Length

This family extends the argument substitution adapter family by adding the ability to replace a reference function argument by the length (as computed by *strlen*) of a target function argument. For instance, the C library function *fwrite* can be adapted to *fputs* by setting its second argument to the constant 1 and its third argument to the length of its first argument. An example where this adapter is useful is shown in Listing 2.3. The first implementation, *isP1*, takes as input a pointer s that points to the beginning of a string, and then computes another pointer p that points to the end of that string using the C function *strlen*. *isP1* then checks that the characters pointed to by s and p are equal as the two pointers are moved closer to each other. The second implementation of the *isPalindrome* predicate, *isP2*, takes as input a pointer to the beginning of a string and that string's length. It then checks that the relevant corresponding characters are equal by indexing into the input string.

```
int isP1(const char *s) {
  const char *p = s + strlen(s) - 1;
  while (s < p)
  if (*p-- != *s++)
    return 0;
  return 1;
}

int isP2(const char *start, size_t len) {
  int i;
  int pal = 1;
  for(i = 0; ((i <= len/2) && (pal==1)); i++) {
    if(start[i] != start[len-i-1])
      pal = 0;
  }
  return pal;
}

int adapted_isP2(const char *s) {
  isP2(s, strlen(s));
}
```

Listing 2.3: Two implementations of the *isPalindrome* predicate

Although *isP1* and *isP2* take a different number of arguments, they implement the same core functionality. *isP1* can be substituted by *isP2* by using an adapter shown in *adapted_isP2*. *adapted_isP2* has the same interface as *isP1* and is implemented using a single call to *isP2* with modified arguments. Notice that it is not possible to adapt the interface of *isP1* to *isP2* because *isP2* checks for the existence of a palindrome *len* bytes long which need not always be equal to the length of the string beginning at *start*.

Argument Substitution with Type Conversion

This adapter family extends the argument substitution adapter family by allowing reference function arguments to be the result of a type conversion applied to a target function argument. Since type information is not available at the binary level, this adapter tries

all possible combinations of type conversion on function arguments. Applying a type conversion at the 64-bit binary level means that each target function argument itself may have been a *char*, *short* or a *int*, thereby using only the low 8, 16, or 32 bits respectively of the argument register. The correct corresponding reference function argument could be produced by either a sign extension or zero extension on the low 8, 16, or 32 bits of the argument register. Listing 2.4 presents an additional adapter that can be synthesized for the target and inner functions in Listing 2.2 when type conversions on target function arguments are allowed during adapter search. The $y\&1$ expression represents a 1-bit zero-extension operation. This adapter family also allows for converting target function arguments to boolean values by comparing those arguments to zero.

```
int adapted_typeconv_f2(int x, unsigned y)
{
  return f2(x,x,x,(y&1));
}
```

Listing 2.4: Type conversion adapter for the function pair shown in Listing 2.2

Arithmetic adapter

This family allows reference function arguments to be arithmetic combinations of target function arguments. To ensure that the space of adapters is finite, our implementation only allows for arithmetic expressions of a specified bounded depth. Since one possible arithmetic combination is adding the constant zero to an argument or to a constant and the arithmetic adapter family allows addition as an operator and includes zero in the set of allowed constants, the arithmetic adapter implicitly allows all the operations allowed under the argument substitution adapter family. Arithmetic adapters allow our tool to reproduce other kinds of synthesis. In general, this adapter family is most useful if a reference function argument is an arithmetic operation over one or more target function arguments and constants. Our adapter synthesis tool can synthesize arithmetic expressions over common binary and unary operators. It can be generalized to supporting operators of higher arity. The primary implementation of this adapter family was done by Hietala [20].

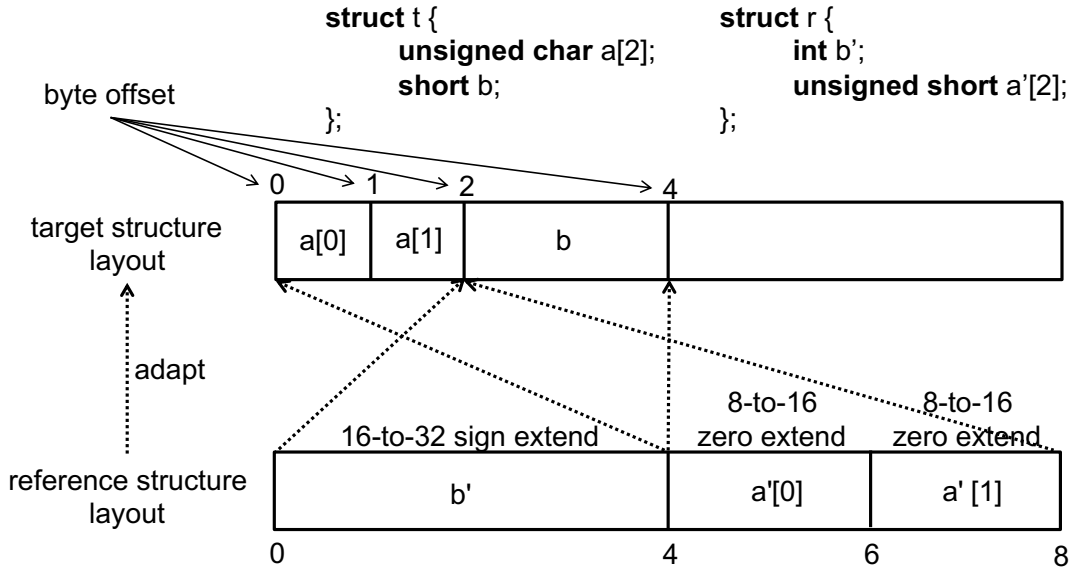
In the description of the capabilities of the synthesis tool SKETCH, Solar-Lezama

et. al. [10] present the synthesis of an efficient bit expression that creates a mask isolating the rightmost 0-bit of an integer. We can synthesize the same bit expression by synthesizing an arithmetic adapter that adapts the identity function to a less-efficient implementation of the operation. In case of the intra-library evaluation done using the C library presented in Section 2.4.5, the arithmetic adapter also allows us to confirm that the binary implementations of *isupper* and *islower* satisfy the assumption that lower-case ASCII characters appear 32 positions higher than their upper-case counterparts in the ASCII table. In other words, the arithmetic adapter allows us to verify an assumption from the binary implementation of *isupper* and *islower*.

Memory Substitution

This adapter family allows a field of a reference function structure argument to be adapted to a field of a target function structure argument. Each field is treated as an array with n entries (where $n \geq 1$), with each entry of size 1, 2, 4, or 8 bytes.

Figure 2.2: Memory substitution adapter to make *struct r* adaptably equivalent to *struct t*



Corresponding array entries used by the target and reference functions need not be

at the same address and may also have different sizes, in which case both sign-extension and zero-extension are valid options to explore for synthesizing the correct adapter as shown in Figure 2.2.

Being able to adaptably substitute inputs and outputs in memory is crucial because it is common for binary code to read inputs from and write outputs to memory. The memory substitution adapter can be used in combination with other adapter families making the overall adapter synthesis more powerful. However, in order to make efficient use of this adapter family, our adapter synthesis tool needs to be told which inputs of the target function are pointers pointing to memory that needs to be adapted. This adapter family synthesizes correct adapters between RC4 implementations in the mbedTLS and OpenSSL libraries in Section 2.4.4.

Return Value Substitution

The argument substitution families described above can also be applied on return values. An example of different return values having the same semantic meaning is the return value of the C library function *isalpha* as shown in Listing 2.1. The wrapper function *adapted_isalpha* in Listing 2.1 performs return value substitution.

Another example demonstrating the need for return value substitution is shown with the pair of functions f_1 and f_2 shown in Listing 2.5. While f_1 returns one if its argument strings are equal and zero otherwise, f_2 returns zero if its argument strings are equal and a non-zero value otherwise. f_1 can be adapted to f_2 only if its return value is changed to accommodate this difference in representation of string equality.

```
int f1(std::string s1, std::string s2) {
  return s1==s2; // compare string objects
}
int f2(std::string s1, std::string s2) {
  return s1.compare(s2); // similar to strcmp
}
int adapted_returnval_f2(std::string s1, std::string s2) {
  return f2(s1,s2)==0;
}
```

Listing 2.5: Two C++ implementations for comparing strings

2.2.3 Example

```
int target(int x, unsigned y) {
  return (x << 1) + (y % 2);
}
int reference(int a, int b, int c, int d) {
  return c + d + (a & b);
}
int adapted_reference(int x, unsigned y) {
  return reference(y, 1, x, x);
}
```

Listing 2.6: Two implementations of similar adaptable functionality

To illustrate our approach, we walk through a representative run of our adapter synthesis algorithm using a pair of target and reference functions that implement similar functionality. Both the target and reference functions are shown in Listing 2.6. Here we will focus only on synthesis of the input adapter, although the general algorithm also produces an adapter that adapts the output of the reference function. A correct input adapter should set the first argument of `reference` to the integer argument y of `target` and set the second argument to the constant 1 to adaptably substitute the `y % 2` operation in `target`. It should also set both the third and fourth arguments of `reference` to the first argument of `target` to adaptably substitute the `x << 1` operation in `target`. We write this adapter as $\mathcal{A}(x, y) = (y, 1, x, x)$.

Step 0: Adapter synthesis begins with an empty counterexample list and a default adapter that maps every argument to the constant 0 (i.e. $\mathcal{A}(x, y) = (0, 0, 0, 0)$). During counterexample generation (`CheckAdapter` in Figure 2.1), we use symbolic execution to search for an inputs x, y such that the output of `target(x,y)` is not equivalent to the output of `reference(`$\mathcal{A}(x, y)$`)` = `reference(0,0,0,0)`. From `CheckAdapter`, we learn that $x = 1, y = 0$ is one such counterexample.

Step 1: Next, during adapter search (`SynthesizeAdapter` in Figure 2.1), we use symbolic execution to search for a new adapter $\mathcal{A}$ that will make `target(x)` equivalent to `reference(`$\mathcal{A}(x, y)$`)` for all inputs x, y in the list [(1,0)]. From `SynthesizeAdapter`, we learn that $\mathcal{A}(x, y) = (y, y, x, x)$ is a suitable adapter, and this becomes our new current adapter.

Step 2: We use `CheckAdapter` to search for a counterexample to the current adapter, $\mathcal{A}(x, y) = (y, y, x, x)$. We find that $x = 0, y = 3$ is a counterexample.

Step 3: Next, we use `SynthesizeAdapter` to search for an adapter $\mathcal{A}$ for which the output of `target(x)` will be equivalent to the output of `reference(`$\mathcal{A}(x, y)$`)` for both pairs of inputs, $x = 1, y = 0$ and $x = 0, y = 3$. `SynthesizeAdapter` identifies $\mathcal{A}(x) = (y, 1, x, x)$ as a suitable adapter.

Step 4: At the beginning of this step, the current adapter is $\mathcal{A}(x, y) = (y, 1, x, x)$. As before, we use `CheckAdapter` to search for a counterexample to the current adapter. We find that `CheckAdapter` fails to find a counterexample for the current adapter, indicating that the current adapter is correct for all explored paths. Therefore, adapter synthesis terminates with the final adapter $\mathcal{A}(x) = (y, 1, x, x)$. Alternatively, adapter synthesis could have terminated with the decision that the target function is not substitutable by the reference function with any allowed adapter. In our evaluations, adapter synthesis may also terminate with a timeout, indicating the total runtime has exceeded a predefined threshold.

2.2.4 Extensibility

The adapter synthesis algorithm presented in this section is not tied to a source programming language or adapter family. In our implementation (Section 2.3) we target binary x86 and ARM code, and we use adapters that allow for common argument structure changes found in binaries compiled from C and C++ code. Because our implementation operates at the binary level, we are also not restricted to working at the level of target functions as described so far. In Section 2.4.7, instead of target functions, we consider target "code fragments." We define a code fragment to be a sequence of instructions consisting of at least one instruction. Inputs to code fragments are the general-purpose registers available on the architecture of the code fragment and outputs are registers written to within the code fragment. We could similarly allow reference functions to be more general code regions. However, we haven't tested our adapter synthesis tool using arbitrary regions of binary code as reference. Allowing arbitrary reference code fragments can potentially allow more useful reference code but it can also make scalability a challenge.

2.3 Implementation

We implement adapter synthesis for Linux/x86-64 binaries using the symbolic execution tool FuzzBALL [21], which is freely available [22]. Symbolic execution determines what inputs to a program will cause certain behaviors. The idea is to execute the program of interest with some concrete values replaced by symbolic variables, and to find satisfying assignments to those symbolic variables that cause the desired execution path to be explored. FuzzBALL allows us to explore execution paths through the target and adapted reference functions to (1) find counterexamples that invalidate previous candidate adapters and (2) find candidate adapters that create behavioral equivalence for the current set of tests. As FuzzBALL symbolically executes a program, it constructs and maintains Vine IR expressions using the BitBlaze [23] Vine library [24] and interfaces with the STP [25] decision procedure to solve path conditions. We also evaluate adapter synthesis by replacing the symbolic execution-based implementation of adapter search with a concrete implementation that searches the adapter space in a random order.

2.3.1 Test Harness

To compare code for equivalence we use a test harness similar to the one used by Ramos et al. [26] to compare C functions for direct equivalence using symbolic execution. The test harness exercises every execution path that passes first through the function, and then through the adapted reference function. As FuzzBALL executes a path through the function, it maintains a path condition that reflects the branches that were taken. As execution proceeds through the adapted reference function on an execution path, FuzzBALL will only take branches that do not contradict the path condition. Thus, symbolic execution through the target and reference functions consistently satisfies the same path condition over the input. Listing 2.7 provides a representative test harness. If the target is a code fragment instead of a function, its inputs x_1, ..., x_n need to be written into the first n general purpose registers available on the architecture. Since the target code fragment may write into the stack pointer register (`sp` on ARM), the value of the stack pointer also needs to be saved before executing the target code fragment and restored after the target code fragment has finished execution. On line 2 the test harness executes `TARGET` with inputs x_1, ..., x_n and captures its output in `r1`. If the

target is a function, its outputs are its return value and values written to memory. If the target is a code fragment, its output needs to be determined in a preprocessing phase. One heuristic for choosing a code fragment's output is to choose the last register that was written into by the code fragment. On line 6, the test harness calls the adapted reference function `REF` with inputs y_1, ..., y_m, which are derived from x_1, ..., x_n using an adapter `A`. After executing `REF`, the test harness adapts `REF`'s return value using the return adapter `R` and saves the adapted return value in `r2`. On line 7 the test harness branches on whether the results of the calls to the target and adapted reference code match.

```
1 void compare(x1, ..., xn) {
2   r1 = TARGET(x1, ..., xn);
3   y1 = adapt(A, x1, ..., xn);
4   ...
5   ym = adapt(A, x1, ..., xn);
6   r2 = adapt(R, REF(y1, ..., ym));
7   if (r1 == r2) printf("Match\n");
8   else printf("Mismatch\n");
9 }
```

Listing 2.7: Test harness

We use the same test harness for both counterexample search (called `CheckAdapter` in Figure 2.1) and adapter search (called `SynthesizeAdapter` in Figure 2.1). During counterexample search, the inputs x_1, ..., x_n are marked as symbolic and the adapters `A` and `R` are concrete. FuzzBALL first executes the function using the symbolic x_1, ..., x_n. It then creates reference function arguments y_1, ..., y_n using the concrete adapter `A` and executes the reference function. During adapter search, for each set of test inputs x_1, ..., x_n, FuzzBALL first executes the function concretely. The adapter `A` is then marked as symbolic, and FuzzBALL then applies symbolic adapter formulas (described in 2.3.2) to the concrete test inputs and passes these symbolic formulas as the adapted reference function arguments y_1, ..., y_n. During counterexample search we are interested in paths that execute the "Mismatch" side, and during adapter search we are interested in paths that execute the "Match" side of the branch on line 7 of Listing 2.7. For simplicity, Listing 2.7 shows only the return values r_1 and r_2 as being used for

equivalence checking. However, during symbolic execution we extend this test harness to do equivalence checking of other state information, including memory and system calls, when comparing the target and adapted reference code executions.

2.3.2 Adapters as Symbolic Formulae

We represent adapter families in FuzzBALL using Vine IR expressions involving symbolic variables. For example, an adapter from the argument substitution family for the adapted reference function argument y_i is represented by a Vine IR expression that indicates whether y_i should be replaced by a constant value (and if so, what constant value) or an argument from the target function (and if so, which argument). This symbolic expression uses two symbolic variables, *y_i_type* and *y_i_val*. We show an example of an adapter from the argument substitution family represented as a symbolic formula in Vine IR in Listing 2.8. This listing assumes the target function takes three arguments, *x1*, *x2*, *x3*. This adapter formula substitutes the adapted reference function argument *y1* with either a constant or with one of the three target function arguments. A value of 1 in *y_1_type* indicates *y1* is to be substituted by the constant value given by *y_1_val*. If *y_1_type* is set to a value other than 1, *y1* is to be substituted by the target function argument at the position present in *y_1_val*. We constrain the range of values *y_1_val* can take by adding side conditions. In the example shown in Listing 2.8, when *y_1_type* equals a value other than 1, *y_1_val* can only equal 0, 1, or 2 since the target function takes 3 arguments.

Symbolic formulas for argument substitution can be extended naturally to more complex adapter families by adding additional symbolic variables. For example, consider the Vine IR formula shown in Listing 2.9 which extends the formula in Listing 2.8 to allow sign extension from the low 16 bits of a value. Listing 2.9 begins in the same way as Listing 2.8 on line 1. But, this time, if *y_1_type* is 0, it performs argument substitution based on the value in *y_1_val* on lines 3, 4. If *y_1_type* is any value other than 0, it performs sign extension of the low 16 bits in a value. This value is chosen based on the position set in *y_1_val* on lines 8, 9. Notice lines 8, 9 are the same as lines 3, 4, which means the value, whose low 16 bits are sign-extended, is chosen in exactly the same way as argument substitution.

```
y_1_type:reg8_t == 1:reg8_t ? y_1_val:reg64_t :
```

```
  ( y_1_val:reg64_t == 0:reg64_t ? x1:reg64_t :
    ( y_1_val:reg64_t == 1:reg64_t ? x2:reg64_t : x3:reg64_t ))
```

Listing 2.8: Argument Substitution adapter

```
1 y_1_type:reg8_t == 1:reg8_t ? y_1_val:reg32_t :
2  ( y_1_type:reg8_t == 0:reg32_t ?
3    ( y_1_val:reg32_t == 0:reg32_t ? x1:reg32_t :
4     ( y_1_val:reg32_t == 1:reg32_t ? x2:reg32_t : x3:reg32_t ))
5   :
6    cast( cast(
7      ( y_1_val:reg32_t == 0:reg32_t ? x1:reg32_t :
8       ( y_1_val:reg32_t == 1:reg32_t ? x2:reg32_t : x3:reg32_t ))
9      L:reg16_t ) S:reg32_t )
```

Listing 2.9: Vine IR formula for one type conversion operation and argument substitution

During adapter search, Vine IR representations of adapted reference function arguments are placed into argument registers of the reference function before it begins execution, and placed into the return value register when the reference function returns to the test harness. When synthesizing memory substitution adapters, Vine IR formulas allowing memory substitutions are written into memory pointed to by reference function arguments. We use the registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9 for function arguments and the register %rax for function return value, as specified by the x86-64 ABI calling convention [27]. We do not currently support synthesizing adapters between functions that use arguments passed on the stack, use variable number of arguments, or specify return values in registers other than %rax.

2.3.3 Equivalence checking of side-effects

As an improvement over previous work [17], in this work, we allow target and reference code to make system calls and have side-effects on memory. We record the side-effects of executing the target and adapted reference functions and compare them for equivalence on every execution path. For equivalence checking of side-effects via system calls, we check the sequence of system calls and their arguments, made by both functions,

for equality. For equivalence checking of side-effects on concretely-addressed memory, we record write operations through both functions and compare the pairs of (address, value) for equivalence. For equivalence checking of side-effects on memory addressed by symbolic values, we use a FuzzBALL feature called *symbolic regions*. Symbolic address expressions encountered during adapted reference function execution are checked for equivalence with those seen during target function execution and mapped to the same symbolic region, if equivalent. To ensure that the target and adapted reference functions both begin execution from the same state of memory, we take a snapshot of memory at the beginning of the target function execution and restore the state of memory using this snapshot before the reference function begins execution.

2.4 Evaluation

In this section, we evaluate the performance of adapter synthesis and present its applications using case studies. In Sections 2.4.1 and 2.4.2, we present case studies that apply adapter synthesis for deobfuscation and finding more efficient clones of a target implementation. In Section 2.4.3, we present an example of finding adaptable substitutability modulo a bug—enabling programmers to more easily replace buggy components of their code. In Section 2.4.4, we show how our technique can enable programmers to switch between different libraries with the same functionality. In Section 2.4.5, we show that adapter synthesis can find adapters even within a library. In Section 2.4.6, we compare symbolic execution-based adapter search with concrete enumeration-based adapter search. Finally, in Section 2.4.7, we show how our technique can be used for reverse engineering.

2.4.1 Case Study: Deobfuscation

A new family of banking malware named Shifu was reported in 2015 [28], [29]. Shifu was found to be targeting banking entities across the UK and Japan. It continues to be updated [30]. Shifu is heavily obfuscated, and one computation used frequently in Shifu is the computation of CRC-32 checksums. We did not have access to the real malware binary, but we were able to simulate its obfuscated checksum computation binary function using freely-available obfuscation tools.

Given a reference implementation of CRC-32 checksum computation, adapter synthesis can be used to check if an obfuscated implementation is adaptably equivalent to the reference implementation. We used the implementation of CRC-32 checksum computation available on the adjunct website [31] of Hacker's Delight [32] (modified so that we could provide the length of the input string) as our reference function. We obfuscated this function at the source code and Intermediate Representation (IR) levels to create three obfuscated clones. For the first clone, we used a tool named Tigress [33] to apply the following source-level obfuscating transformations:

1. Function virtualization: This transformation turns the reference function into an interpreter with its own bytecode language.

2. Just-in-time compilation/dynamic unpacking: This transformation translates each function f into function f' consisting of intermediate code so that, when f' is executed, f is dynamically compiled to machine code.

3. reordering the function arguments randomly, inserting bogus arguments, adding bogus non-trivial functions and loops, and allowing superoperators [34].

These transformations led to a 250% increase in the number of source code lines. For the second clone, we applied the following obfuscating transformations at the LLVM IR level using Obfuscator-LLVM [35]:

1. Instruction Substitution: This transformation replaces standard binary operators like addition, subtraction, and boolean operators with functionally equivalent, but more complicated, instruction sequences.

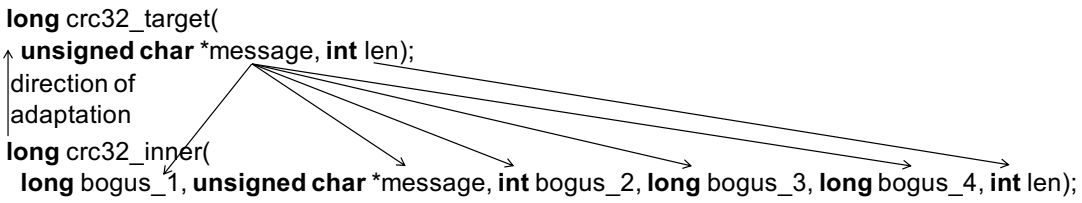
2. Bogus Control Flow: This transformation modifies the function call graph by injecting basic blocks for bogus control flow and modifying existing basic blocks by adding junk instructions chosen at random.

3. Control flow flattening: This transformation flattens the control flow graph of the clone in a way similar to László et al [36].

These transformations caused the number of instruction bytes to increase from 126 to 2944 bytes. Finally, we compiled the obfuscated C code (obtained using Tigress) with

the LLVM obfuscator tool to create a third clone. We then ran our adapter synthesis tool with the reference function as the target function and all three clones as inner functions. We used the CRC-32 checksum of a symbolic 1 byte input string as the return value of each clone. Our adapter synthesis tool, using FuzzBALL-based adapter search, correctly concluded that all three clones were adaptably equivalent to the reference function in less than 3 minutes using argument substitution. A correct adapter for one obfuscated clone is shown in Figure 2.3. It maps the string and length arguments correctly, while ignoring the four bogus arguments (the mappings to bogus arguments are irrelevant). While performing adapter synthesis on heavily-obfuscated binary code is challenging, adaptation in this example is complicated further by an increase in the number of inner function arguments causing the adapter search space to increase to 43.68 million adapters.

Figure 2.3: Argument substitution adapter to make one obfuscated CRC-32 checksum function adaptably equivalent to the reference function



2.4.2 Case Study: Efficiency

Adapter synthesis can also be applied to find more efficient versions of a function, even when those versions have a different interface. Matrix multiplication is one of the most frequently-used operations in mathematics and computer science. It can be used for other crucial matrix operations (for example, gaussian elimination, LU decomposition [37]) and as a subroutine in other fast algorithms (for example, for graph transitive closure). Adapting faster binary implementations of matrix multiplication to the naive one's interface improves the runtime of such other operations relying on matrix multiplication. Hence, as our target function, we use the naive implementation of matrix multiplication shown in Listing 2.10. As our inner function we use an implementation of Strassen's algorithm [38] from Intel's website [39], which takes the input matrices A

and B as the 1st and 2nd arguments respectively and places the product matrix in its 3rd argument. We modified their implementation so that it used Strassen's algorithm for all matrix sizes. Our adapter synthesis tool, using FuzzBALL-based adapter search, finds the correct argument substitution adapter for making the implementation using Strassen's algorithm adaptably equivalent to the naive implementation in 17.7 minutes for matrices of size 4x4. When using concrete enumeration-based adapter search, the adapter search finds the correct adapter in less than 4.5 minutes.

```
#define N 4
void naive_mm(int** C, int** A, int** B) {
  int i, j, k;
  for ( i = 0; i < N; i++)
    for ( j = 0; j < N; j++) {
      C[i][j] = 0;
      for ( k = 0; k < N; k++)
        C[i][j] += A[i][k]*B[k][j];
    }
}
```

Listing 2.10: Naive implementation of matrix multiplication

This example shows that adapter synthesis can be used for finding adaptably equivalent clones of a function that have different algorithmic space and time complexity. Program analysis techniques for checking space and time usage of different implementations are being actively researched [40]. Symbolic execution can also be used for finding inputs that cause functions to exhibit worst-case computational complexity [41]. adapter synthesis can be used as a pre-processing step before applying other techniques for detecting algorithmic complexity of semantic clones.

2.4.3 Case Study: Security

Data structures that map keys to their corresponding values are commonly found in modern programming languages [42], [43]. Support for such data structures is not part of the C language standard. Arrays in C provide a convenient and fast way for mapping a previously known number of integer keys to values. Implementation of data compression algorithms and signal processing implementations require constant-time lookup for

negative and positive powers of values. In such situations, an array in C can be the perfect solution. Consider a array-based table implementing a function of a signed input. For example, keys ranging from -127 to 127 can be mapped to a 255-element array. Any key k will then be mapped to the element at position k+127 in this array. We present two implementations of such lookup functions in Listing 2.11. Both functions, *lookup1* and *lookup2*, assume keys ranging from $-len/2$ to $+len/2$ are mapped in the *table* parameter with *lookup1* being specific to tables of length 255. However, *lookup1* contains a bug caused by undefined behavior. The return value of *abs* for the most negative 32-bit integer (-2147483648) is not defined [44]. Given the most negative 32-bit integer, the eglibc-2.19 implementation of *abs* returns that same 32-bit integer. This causes the check on line 2 of Listing 2.11 to not be satisfied, allowing execution to continue to line 5 and causing an out-of-bounds memory access. *lookup2* in Listing 2.11 is a different implementation of an array-based lookup with a different interface than *lookup1*. Checking whether the key is in range is done differently in *lookup2*, causing it to not have the memory access bug present in *lookup1*. Users of *lookup1* may find it desirable to substitute the use of *lookup1* with *lookup2*. Adapter synthesis can perform such a substitution by adapting *lookup2* to *lookup1* while simultaneously not adapting the out-of-bounds memory access in *lookup1*. Given a mechanism to detect when such a out-of-bounds memory access happens in the target function and the reference function *lookup2*, our adapter synthesis implementation synthesizes the correct argument substitution adapter in the *lookup1* $\leftarrow$ *lookup2* direction in about 8 minutes.

For this case study, we provided a predicate to the adapter synthesis tool that, when violated, indicates a out-of-bounds memory access in the target function, *lookup1*. Any execution path on which the out-of-bounds memory access was detected in the target function was terminated early within the target function itself and not allowed to continue to executing the adapted reference function. This modification to adapter synthesis caused the reference function to not be required to also have the out-of-bounds memory access. While we did not automate detection of such out-of-bounds memory accesses for arbitrary binary code, integrating such a feature can allow use of adapter synthesis for binary program repair. For the purposes of this case study, we set the adapter synthesis up with only a single reference function. We plan to explore the application of adapter synthesis for binary program repair in the future by running a

larger evaluation where we attempt to synthesize adapters using other reference functions. Synthesis of the correct adapter is slowed down by the presence of the *table* pointer in the interfaces of *lookup1* and *lookup2*. The adapter is shown on lines 15-18 of Listing 2.11. This case study shows adapter synthesis can replace a buggy function with its bug-free variant by doing adaptation modulo a bug.

```
1  unsigned char lookup1
2      (unsigned char *table, int key) {
3    if(abs(key) > 127) //buggy for key = -2147483648
4      return -1;
5    return table[key+127];
6  }
7
8  unsigned char lookup2
9      (unsigned char *table, int len, int key) {
10   if( !(-(len/2) <= key && key <= (len/2)) )
11     return -1;
12   return table[key+(len/2)];
13 }
14
15 unsigned char adapted_lookup2
16     (unsigned char *table, int key) {
17   return lookup2 (table, 255, key);
18 }
```

Listing 2.11: Two implementations for mapping ordered keys, negative or positive, to values using a C array

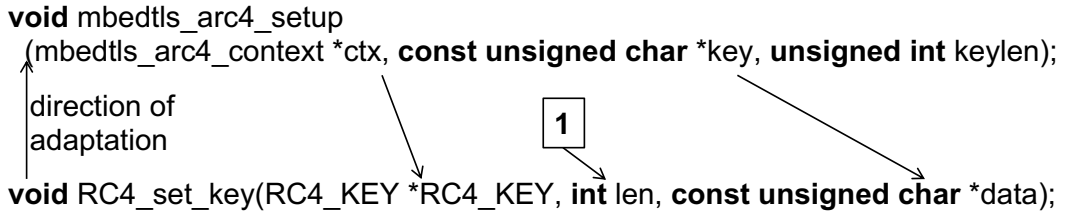
2.4.4 Case Study: Library Replacement

To show that adapter synthesis can be applied to replace functionality from one library with that from another library, we adapt functions implementing RC4 functionality in mbedTLS and OpenSSL.

RC4 context initialization

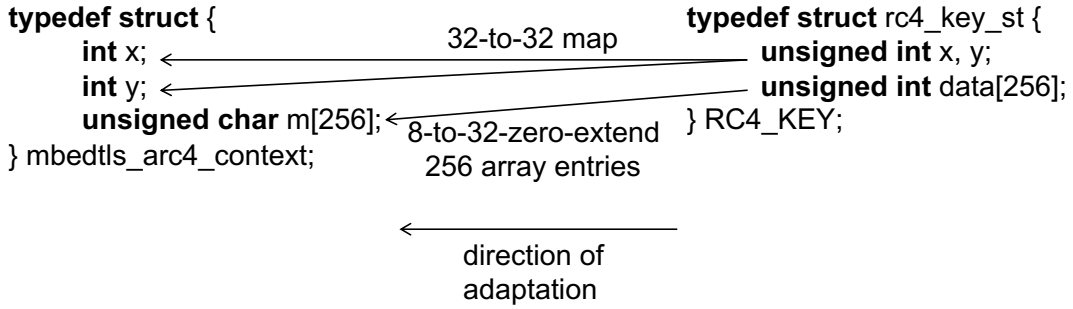
The RC4 algorithm uses a variable length input key to initialize a table with 256 entries within the context argument. Both cryptography libraries in our example, mbedTLS and OpenSSL, have their own implementation of this initialization routine. Both initialization function signatures are shown in Figure 2.4.

Figure 2.4: Argument substitution adapter to make *RC4_set_key* adaptably equivalent to *mbedtls_arc4_setup*



Executing each of these initialization routines requires 256 rounds of mixing bytes from the key string into the context. Each round requires two load and two store operations into an array with 256 entries. The two initialization routines require the key length and key string arguments at different positions, and use different RC4 context structures (*RC4_KEY* in OpenSSL, *mbedtls_arc4_context* in mbedTLS). The RC4 context arguments contain three fields as shown in Figure 2.5.

Figure 2.5: Memory substitution adapter to make *RC4_KEY* adaptably equivalent to *mbedtls_arc4_context*



The first two 4-byte fields are used to index into the third field, which is an array with 256 entries. Each entry in the array is 4 bytes wide in OpenSSL and 1 byte wide in mbedTLS. The correct adapter that adapts the OpenSSL context to the mbedTLS context (mbedTLS ← OpenSSL) performs two mapping operations: (1) it maps the first two mbedTLS context's fields directly to the first two OpenSSL context's fields and (2) it zero extends each 1 byte entry in the third field of the mbedTLS context to the corresponding 4 byte entry in the third field of the OpenSSL context. The correct adapter for making the *RC4_KEY* structure adaptably equivalent to the *mbedtls_arc4_context* structure is shown in Figure 2.5. The correct adapter in the reverse direction (OpenSSL ← mbedTLS) changes the second mapping operation to map the least significant byte of each 4 byte entry in the third field to the 1 byte entry in its corresponding position. In our library replacement case study, these two types of integers in the key structures are only ever used to hold a 1-byte value. This phenomenon coincidentally makes these RC4 implementations adaptable. It should be noted that it may be correct for an adapter to map unsigned integers to signed integers and vice-versa. Even though these two types have two different ranges of values, an adapter does not need to be value-preserving to be correct.

Performing this adaptation with *mbedtls_arc4_setup* and *RC4_set_key* (the RC4 context initialization functions in mbedTLS and OpenSSL respectively) requires adaptation of side-effects on memory because mixing of the key string into the context is the only output of these functions. Since a memory substitution structure can be used both as input and as output, we have to perform the memory substitution adaptation at least twice. First, the reference function may use the memory substitution structure as input. Hence, we need to adapt the initial byte values of the memory substitution structure to obtain the initial byte values to be used for the reference function. Second, before running the reference function, the target function could have written to the memory substitution structure. Hence, we need to adapt side-effects of the target function on the memory substitution structure in order to compare them with corresponding side-effects on memory from the reference function. The most general memory substitution adapter synthesis allows arbitrary numbers of 1, 2, 4, or 8 byte entries in each field of the 264 ($2\times4+256\times1$) byte mbedTLS context and 1032 ($2\times4+256\times4$) byte OpenSSL context. But this makes the search space of memory mappings very large. We instead

only explored adapters where the number of entries in each array was a power of 2. While this reduction is useful in practice, it still gives us a search space of about 4.7 million possible memory substitutions in both directions of adaptation.

Finally, memory substitution must be combined with argument substitution to synthesize adapters between *mbedtls_arc4_setup* and *RC4_set_key*. This combination of argument substitution and memory substitution adapter families creates a search space of 5.593 billion adapters. Our adapter synthesis tool figures out the correct argument, memory, and return value substitutions. It finds adaptable equivalence in both directions of adaptation by checking equivalence between side-effects on the structure objects (*ctx* for *mbedtls_arc4_setup*, *RC4_KEY* for *RC4_set_key*). The correct adapter for adaptably substituting the *mbedtls_arc4_setup* function with the *RC4_set_key* function is shown in Figure 2.4. To setup adapter synthesis between these two function pairs (we synthesized adapters in both directions), we used a symbolic key string of length 1, and hence the synthesis tool correctly sets the key length argument to 1. While we acknowledge that using an input string of length 1 is too small to be useful, we expect the adapter to be correct on strings of length greater than 1 in practice. We also plan to integrate techniques such as path merging [4,5] to increase the bounds of inputs used in adapter synthesis. While we used an input memory substitution size of 1032 symbolic bytes for memory substitution, both *mbedtls_arc4_setup* and *RC4_set_key* initialize this memory with concrete values in their implementation, thereby causing this adaptation to start with a much smaller symbolic state consisting of a single symbolic input byte.

We present the time taken to synthesize adapters for RC4 setup function pairs in Table 2.1. The execution time shown in Table 2.1 for concrete enumeration-based adapter search is the average execution time taken for adapter synthesis over 10 correctly synthesized adapters for adapting RC4 setup functions. We performed adapter synthesis using concrete enumeration-based adapter search 10 times because concrete enumeration-based adapter search traverses the adapter space in a random order. Using the execution times shown in Table 2.1 and the observation that our adapter synthesis never used more than 1 CPU core and 4 GB of RAM, we estimated [45] the cost of this computation on a Amazon EC2 instance (t2-medium). Table 2.2 shows these estimated

costs. These costs suggest that automated adapter search is likely competitive with paying a human programmer to find and verify the correctness of an adapter at the binary level. The time required for adapter synthesis can be further reduced by parallelizing the adapter search in concrete enumeration and reusing the state of adapter search in FuzzBALL-based adapter search. We discuss this further in Section 2.5.

Table 2.1: Time taken in days for synthesizing adapters between RC4 setup and encryption functions in OpenSSL and mbedTLS

	setup (M ← O)	setup (O ← M)	enc (M ← O)	enc (O ← M)
Concrete enumeration-based adapter search	8.24	5.52	5.65	5.52
FuzzBALL-based adapter search	7.87	6.15	8.54	2.10

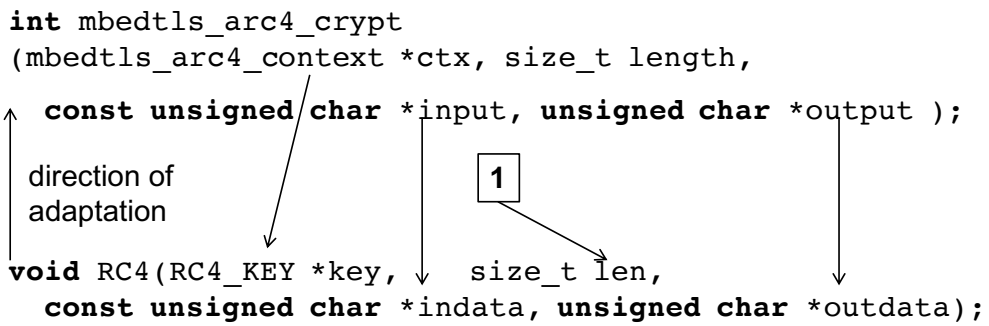
Table 2.2: Estimated cost (in USD) of synthesizing adapters between RC4 setup and encryption functions in OpenSSL and mbedTLS on an Amazon EC2 instance

	setup M ← O	setup O ← M	enc M ← O	enc O ← M	Total
Concrete enum.-based adapter search	$9.17	$6.15	$6.29	$6.15	$27.76
FuzzBALL-based adapter search	$8.76	$6.84	$9.51	$2.34	$27.45

RC4 encryption

The RC4 encryption functions in mbedTLS and OpenSSL take 4 arguments each, three of which are pointers to the RC4 context, the input key string, and the output string, as shown in Figure 2.6.

Figure 2.6: Argument and memory substitution adapters to make *mbedtls_crypt* in the mbedTLS library adaptably substitutable by *RC4* in OpenSSL



These functions use the RC4 context as input, causing the initial symbolic state to consist of 1032 symbolic bytes for memory substitution and one symbolic byte for the input string. These functions both read from and write to the RC4 context, making two memory substitution adaptations necessary. These functions also encrypt every byte of the input string using a loop where the length of the input string is given as a parameter to the function. Since all arguments to the reference function are symbolic, using a symbolic formula for the length of the input string can easily cause the loop bound to be very large, especially if the symbolic formula for the length allows the possibility of the length being equal to one of the pointer arguments to the reference function. To avoid the encryption loop in the reference function from executing too many times, we restricted every instruction in the reference function to be executed at most twice. Finally, because these functions write to an output string, it is necessary for us to have memory side-effects equivalence checking to capture outputs that are not part of the return value.

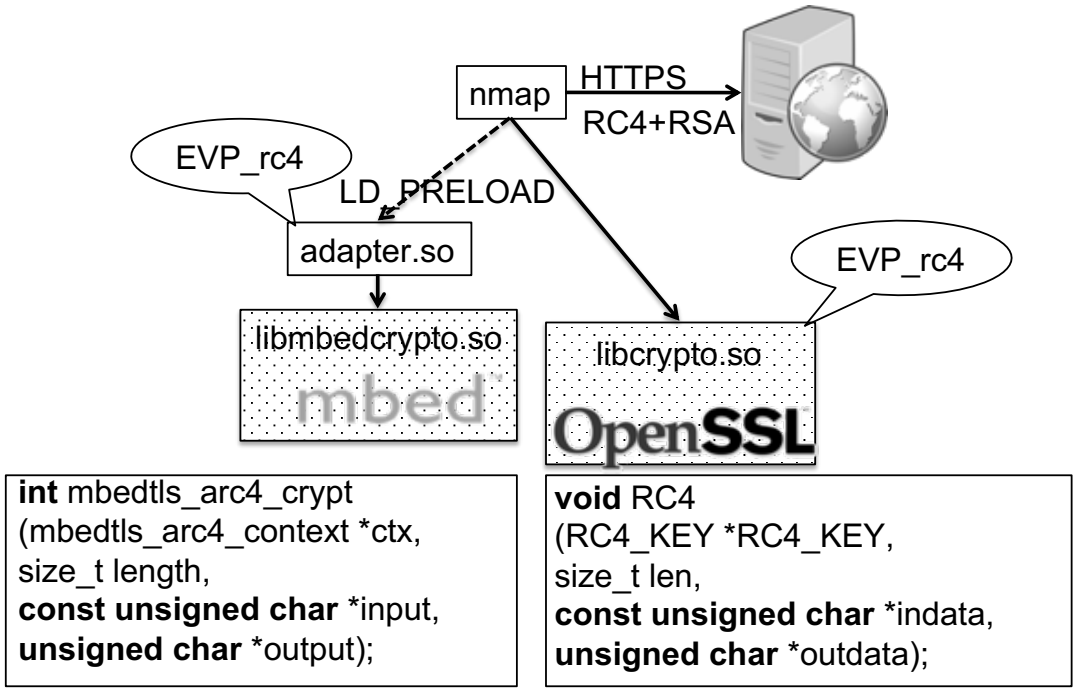
The adapter search space in this case consists of 1.792×10^{12} adapters. The correct adapter for making the *RC4* method adaptably equivalent to *mbedtls_arc4_crypt* is

shown in Figure 2.6. Our adapter synthesis tool finds the correct argument and memory substitution adapters in both directions of adaptation. Tables 2.1, 2.2 show the time taken for and estimated cost of adapter synthesis between the RC4 encryption functions in OpenSSL and mbedTLS. Once again, the execution time shown in Table 2.1 for concrete enumeration-based adapter search is the average execution time taken for adapter synthesis over 10 correctly synthesized adapters. We verified the correctness of our adapted context structures by using self-tests present in mbedTLS and OpenSSL.

RC4 adapter verification using nmap

We verified the correctness of our RC4 memory substitution adapter using nmap with the setup shown in Figure 2.7.

Figure 2.7: nmap using RC4 encryption in mbedTLS instead of OpenSSL



We created adapted versions of the OpenSSL RC4 setup and encryption functions that internally use the mbedTLS context adapted to the OpenSSL context. On a 64-bit virtual machine running Ubuntu 14.04, we compiled the adapted setup and encryption

functions into a shared library and setup a local webserver, which communicated over port 443 using the *RC4+RSA* cipher. We used the stock nmap binary to scan our localhost and injected our specially created shared library using the *LD_PRELOAD* environment variable. The preloading caused the RC4 functions in our shared library to be executed instead of the ones inside OpenSSL. The output of nmap, run with preloading our specially created shared library which used the OpenSSL ← mbedTLS structure adapter, was the same as the output of nmap when using the system OpenSSL library.

2.4.5 Intra-library Adapter Synthesis

The previous section showed an application of adapter synthesis where the target and reference functions came from independently-developed implementations. But, adapter synthesis can also be useful in cases where the target and reference functions were developed within the same library. Synthesizing adapters between binary functions in the same library can expose important relations between adaptably substitutable functions that may not be known to users of the library. It can show relations between functions, by, for example showing that one function can be adaptably implemented in terms of another. Verifying such relations between functions from their binary implementation can provide the users of the library a more detailed picture of the function's behavior. In libraries with a large interface, such as the Ubuntu 14.04 system C library, where it can be challenging to manually identify adaptability relations between functions, performing automated intra-library adapter synthesis can be particularly useful.

Setup

We evaluated our adapter synthesis tool on the system C library available on Ubuntu 14.04 (eglibc 2.19). The C library uses a significant amount of inlined assembly, for instance, the *ffs*, *ffsl*, *ffsll* functions, which motivates automated adapter synthesis at the binary level. We enumerated 1316 exported functions in the library in the order they appear, which caused functions that are defined in the same source files to appear close to each other. Considering every function in this list as the target function, we chose five functions that appear above and below it as 10 potential reference functions. These steps gave us a list of 13130 ($10 \times 1316 - 2 \times \sum_{i=1}^{5} i$) pairs of target and reference

functions. We used the argument substitution, type conversion, and arithmetic adapter families combined with the return value adapter family because these families scale well and are widely applicable. We ran our adapter synthesis on a machine running Ubuntu 16.04 with 64-bit Linux kernel version 4.15.0-45-generic using 192 GB RAM and a Intel(R) Xeon(R) CPU E5-2623 v3 processor. We used a 5 minute time limit for each adapter synthesis task using the argument substitution and type conversion adapters and a 10 minute time limit for tasks using arithmetic adapter. This extra time was required for the arithmetic adapter because we observed the solver took longer to solve arithmetic adapter formulas during the adapter search step. To keep the running time of the entire adapter synthesis process within practical limits, we configured FuzzBALL to use a 5 second SMT solver timeout and to consider any queries that trigger this timeout as unsatisfiable. We limited the maximum number of times any instruction can be executed to 4000 because this allowed execution of code which loaded library dependencies. We limited memory regions to be symbolic up to a 936 byte offset limit (the size of the largest structure in the library interface) and any offset outside this range was considered to contain zero.

We also combined the memory substitution adapter family with argument substitution and return value substitution to check if we could find any useful memory substitutions within the C library. For evaluating the use of the memory substitution adapter, we constructed a list of 69 C library functions that, among all their arguments, take exactly one pointer argument. Our memory substitution adapter synthesis depends on knowledge of which of the target arguments is a pointer to a structure that needs to be adapted. This is a limitation that we plan to alleviate in the future. While the memory substitution adapter can also be synthesized for an arbitrarily large structure size and an arbitrarily large number of fields, the memory substitution adapter search can be optimized by giving it a tight bound on both of these parameters. Therefore, we plugged in knowledge of the size of each structure and the number of fields in the structure to the adapter synthesis process. We used a 10 minute time limit in this evaluation too because we observed that the memory substitution adapter search process is slow when compared to other adapter families. We set up these four adapter synthesis evaluations up to begin with a default “zero” adapter that set all the reference function arguments to the constant 0.

Results

Table 2.3 summarizes the results of searching for argument substitution, type conversion, and arithmetic adapters with a return value adapter within the 13130 function pairs described above. The final row shows the results of combining memory substitution with argument substitution and return value substitution over 4692 (69*68) function pairs within the C library. The similarity in the results for the type conversion adapter family and argument substitution adapter family arises from the similarity of these two families. The most common causes of crashing during execution of the target function were missing system call support in FuzzBALL and incorrect null dereferences (caused due to lack of proper initialization of pointer arguments). The timeout column includes all function pairs for which we had a solver timeout (5 seconds), hit the iteration limit (4000), or reached a hard timeout (2 minutes). The search terminated without a timeout for 70% of the function pairs, which reflects a complete exploration of the space of symbolic inputs to a function, or of adapters. The proportion of timeouts among the total number of adapter synthesis tasks when using the memory substitution adapter is much higher due to the memory substitution adapter being slow to synthesize.

Table 2.3: Adapter Synthesis over 13130 function pairs using 3 adapter setups. The final row shows an evaluation done using all pairs of target and reference functions chosen from a list of 69 Glibc functions that each take exactly one pointer argument. The total number of function pairs in the final row is 4692 (69*68).

adapter type	no adapter #	adapters found #	timeout #	target function crashed #
arg.sub.+ ret.val.sub.	9942	404	1801	983
type.conv.+ ret.val.sub	9990	414	1840	986
arithmetic+ ret.val.sub	9661	377	2203	889
arg.sub+ memory sub.+ ret.val.sub	1263	0	3090	339

Table 2.4: Adapters found within eglibc-2.19 using the type conversion, arithmetic adapter, and return value substitution adapter families. `f1` ← `f2` represents `f1`, the target function, being adaptably substituted by `f2`, the reference function. `32-to-64S` represents 32-to-64 bit sign extension. `32-to-64Z` represents 32-to-64 bit unsigned extension. `K` represents a constant value and `#K` represents the argument at position `K`.

$f_1 \leftarrow f_2$ or $f_1 \leftrightarrow f_2$	adapter
abs(1) ← labs(1) abs(1) ← llabs(1)	32-to-64S(#0) and 32-to-64Z(return value)
labs(1) ↔ llabs(1)	#0
ffs(1) ← ffsl(1) ffs(1) ← ffsll(1)	32-to-64S(#0)
ffsl(1) ↔ ffsll(1)	#0
setpgrp(0) ← setpgid(2)	0, 0
wait(1) ← waitpid(3)	-1, #0, 0
wait(1) ← wait4(4)	-1, #0, 0, 0
waitpid(3) ← wait4(4)	#0, #1, #2, 0
wait(1) ← wait3(3)	#0, 0, 0
wait3(3) ← wait4(4)	-1, #0, #1, #2
umount(1) ← umount2(2)	#0, 0
ldiv(1) ↔ lldiv(1)	#0
recv(4) ← recvfrom(6) send(4) ← sendto(6)	32-to-64S(#0), #1, #2, 32-to-64S(#3), 0, 0
toascii(1) ← isascii(1)	#0 & 0x7f, return value=#0 & 0x7f
atol(1) ↔ atoll(1)	#0
atol(1) ← strtol(3) atoi(1) ← strtol(3) atoll(1) ← strtoll(3)	#0, 0, 10
isupper(1) ← islower(1)	#0 + 32
islower(1) ← isupper(1)	#0 - 32
killpg(1) ← kill(1)	-#0, #1

Since there is no ground truth, we manually corroborated the results of our evaluation by checking the C library documentation and source code. Our adapter synthesis evaluation on the C library reported 28 interesting true positives shown in Table 2.4. The remaining adapters found were correct, but trivial. We did not find any interesting results among synthesized adapters using the memory substitution adapter family. The arithmetic adapter family did report the same adapters that were reported by the argument substitution family since the arithmetic adapter also allows simple argument substitution. One interesting case of arithmetic adaptation was the adaptable substitution of `toascii` using `isascii`. The `toascii` method converts a value to ASCII by taking a bitwise and with the constant `0x7f`. The arithmetic adaptation of the first argument to `isascii` causes `isascii` to be called with a value that is a bitwise and of its original argument with the constant `0x7f`. Effectively, the arithmetic adaptation is sophisticated enough to subsume the functionality of `toascii`. All the other adapters reported in Table 2.4 were found by the argument substitution and type conversion families by combining them with return value substitution. The first column in Table 2.4 shows the function pair between which an adapter was found (with the number of arguments) and the second column shows the adapter. We use the following notation to describe adapters in a compact way. $f_1 \leftrightarrow f_2$ means $f_1 \leftarrow f_2$ and $f_2 \leftarrow f_1$. # followed by a number indicates reference argument substitution by a target argument, while other numbers indicate constants. X-to-YS represents taking the low X bits and sign extending them to Y bits, X-to-YZ represents a similar operation using zero extension.

The last five rows in Table 2.4 show adapters that were synthesized with some manual intervention.

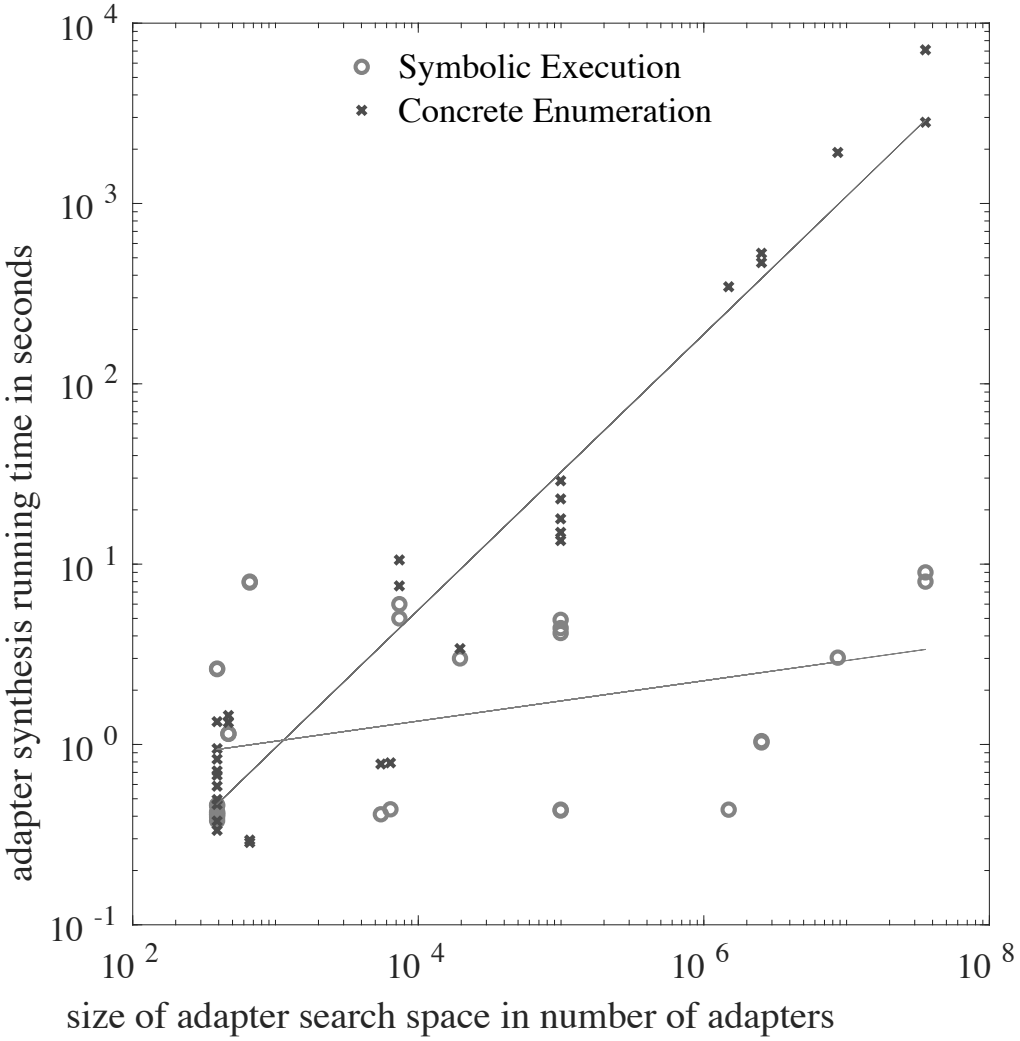
- The adapters synthesized using `atol`, `atoll`, and `atoi` as the target functions involved using a 15 minute timeout with argument substitution because the final counter-example search step for the correct adapter takes about 14 minutes.

- The last three rows shown in Table 2.4 shows three arithmetic adapters found within the C library using partial automation. The functions *isupper*, *islower*, *kill* have assumptions about conditions that will be satisifed by inputs given to them. We synthesized the correct adapters by writing wrappers containing preconditions around these three functions.

2.4.6 Comparison with Concrete Enumeration-based Adapter Search

For every adapter family that we have discussed, the space of possible adapters is finite. So instead of using symbolic execution for adapter search, we can find candidate adapters by enumerating concrete adapters until we find one that produces equal side-effects and return values for all previously-found tests. We implement concrete enumeration-based adapter search in C for all the adapter families described in Section 2.2. We use the Pin [46] framework for checking side-effects on memory and system calls for equality. To prevent adapter search time from depending on the order of enumeration, we randomize the sequence in which adapters are generated, ensuring that every adapter had the same probability of being generated. For the adaptable function pairs reported in Section 2.4.5, we synthesized adapters from the type conversion adapter family using both the concrete enumeration- and symbolic execution-based adapter search implementations and captured the total adapter search time. We also counted the size of the adapter search space for every adaptation. In some cases, the adapter search space was too large to be concretely enumerated. For example, the adapter search space for the *killpg* $\leftarrow$ *kill* adapter consists of 98.1 million arithmetic adapters. In such cases, we reduced the size of the search space by using smaller constant bounds. Based on the size of adapter search space, we compared the total adapter search times for both adapter search strategies. We present the results from this comparison in Figure 2.8.

Figure 2.8: Comparing concrete enumeration-based adapter search with binary symbolic execution-based adapter search for adapters presented in Section 2.4.5. This figure shows that, with an increase in adapter search space, the total runtime of adapter synthesis increases exponentially for concrete enumeration-based adapter search., whereas the increase in adapter search space has little effect on symbolic execution-based adapter search.



For concrete enumeration-based adapter search, Figure 2.8 shows the time required

to find an adapter has a consistent exponential increase with an increase in the size of adapter search space. But when using binary symbolic execution-based adapter search, Figure 2.8 shows a much slower increase in time required to find the adapter. This occurs because symbolic execution is less affected by an increase in the size of the adapter search space due to an increase in the number of arguments and the number of possible constants in the adapter family.

2.4.7 Large-Scale Reverse Engineering

In this section, we show how adapter synthesis can be used for reverse engineering. Our goal is to understand the behavior of fragments of binary code by synthesizing adapters between those fragments and reference functions with known behavior. We evaluate on binary code fragments taken from an ARM firmware image and reference functions chosen from the source code of a popular media player.

Code fragment selection

Rockbox [47] is a free replacement 3rd party firmware for digital music players. We used a Rockbox image compiled for the iPod Nano (2g) device, based on the 32-bit ARM architecture, and disassembled it. We dissected the firmware image into code fragments using the following rules: (1) no code fragment could use memory, stack, floating-point, coprocessor, or supervisor call instructions, (2) no code fragment could branch to an address outside itself, (3) the first instruction of a code fragment could not be conditionally executed.

The first rule disallowed code fragments from having any inputs from/outputs to memory, thereby allowing us to use the 13 general purpose registers on ARM as inputs. The second rule prevented a branch to an invalid address. ARM instructions can be executed based on a condition code specified in the instruction. If the condition is not satisfied, the instruction is turned into a `noop`. The third rule disallowed the possibility of having code fragments that begin with a `noop` instruction, or whose behavior depends on a condition. The outputs of every code fragment were the last (up to) three registers written to by the code fragment. This caused each code fragment to be used as the target code region up to three times, once for each output register. This procedure gave

us a total of 183,653 code regions, with 61,680 of them consisting of between 3 and 20 ARM instructions.

To evaluate which code fragments could be synthesized just with our adapter family without a contribution from a reference function, we checked which of these 61,680 code fragments could be adaptably substituted by a reference function that simply returns one of its arguments. We call this reference function an "identity" function and assume it is uninteresting to identify target fragments that simply return one of their arguments. Intuitively, any code fragment that can be adaptably substituted by such an uninteresting reference function must be uninteresting itself, and so need not be considered further. We found 46,831 of the 61,680 code fragments could not be adaptably substituted by the identity reference function, and so we focused our further evaluation on these 46,831 code fragments that were between 3 and 20 ARM instructions long and non-trivial.

Reference functions

Since our code fragments consisted of between 3 and 20 ARM instructions, we focused on using reference functions that can be expressed in a similar number of ARM instructions. We used the source code of version 2.2.6 of the VLC media player [19] as the codebase for our reference functions. We performed an automated search for functions that were up to 20 lines of source code. This step gave us a total of 1647 functions. Similar to the three rules for code fragment selection, we discarded functions that accessed memory, called other VLC-specific functions, or made system calls to find 24 reference functions. Other than coming from a broadly similar problem domain (media players), our selection of reference functions was independent of the Rockbox codebase, so we would not expect that every reference function would be found in Rockbox.

Results

We used the type conversion adapter family along with the return value substitution family, disallowing return value substitution adapters from setting the return value to be a type-converted argument of the reference function (which would lead to uninteresting adapters). We allowed the reference function arguments to be replaced by unrestricted

32-bit constants, and we assumed each code segment takes up to 13 arguments. The size of this adapter search space can be calculated using the following formula:

$$8 \times \sum_{k=0}^{k=13} (2^{32})^{13-k} \times {}^{13}C_k \times {}^{13}P_k \times 8^k$$

The first multiplicative factor of 8 is due to the 8 possible return value substitution adapters. The permutation and combination operators occur due to the choices of arguments for the target code fragment and reference functions (we assumed both have 13 arguments since most general-purpose registers can be used as input in an arbitrary code fragment). The final 8^k represents the 8 possible type conversion operators that a type conversion adapter can apply. The dominant factor for the size of the adapter search space comes from the size of the set of possible constants. Our adapter family used unrestricted 32-bit constants, leading to a constants set of size 2^{32}.

With this adapter family set up, we ran adapter synthesis trying to adaptably substitute each of the 46,831 code fragments by each reference function. This gave us a total of 1,123,944 (46831×24) adapter synthesis tasks, with each type conversion and return value substitution adapter synthesis search space consisting of 1.353×10^{127} adapters, too large for concrete enumeration. We also used this setup for evaluating the use of the arithmetic substitution adapter family along with return value substutition. The arithmetic adapter was allowed to use arithmetic combinations of arguments and constants using the addition and subtraction binary operators. We set a 10 minute hard time limit and a total memory limit of 2 GB per adapter synthesis task for the type conversion with return value substitution adapter synthesis and for arithmetic adapter synthesis with return value substitution. We split the adapter synthesis tasks for each reference function into 32 parallel jobs, creating a total of 768 (32×24) parallel jobs. We ran our jobs on a machine cluster running CentOS 6.8 with 64-bit Linux kernel version 2.6.32 and Intel Xeon E5-2680v3 processors.

Table 2.5: Reverse engineering results using type conversion substitution adapter for constructing the reference function arguments: using 46831 target code fragments from a Rockbox firmware image and the first 12 of 24 reference functions from VLC media player grouped by the three overall possible terminations of adapter synthesis. The *#(full)* column reports how many code fragments were found to be adaptably substitutable, and how many of those exploited the full generality of the reference function.

function name	adapter found				no adapter	timeout
	# (full)	#cluster	steps	total time	#	#
clamp	2171 (877)	338	8.9	51.7	43496	1156
prev_ pow_2	912 (0)	15	3.7	7.5	45889	22
abs_diff	1511 (56)	88	6.4	14.9	45284	28
bswap32	999 (8)	29	4.6	8.0	45801	23
integer_ cmp	1025 (5)	27	4.8	10.4	45628	170
even	892 (1)	10	3.9	7.2	45910	21
div255	884 (0)	16	3.7	6.3	45917	22
reverse_bits	1179 (32)	23	4.8	11.1	45621	23
binary_log	963 (0)	10	3.8	8.0	45654	206
median	1601 (319)	202	7.5	47.8	41300	3922
hex_value	880 (0)	7	3.6	6.5	45632	311
get_descr_ len_24b	916 (0)	11	4.5	7.9	45596	311

Table 2.6: Reverse engineering results using type conversion substitution adapter for constructing the reference function arguments: using 46831 target code fragments from a Rockbox firmware image and the second 12 of 24 reference functions from VLC media player grouped by the three overall possible terminations of adapter synthesis. The *#(full)* column reports how many code fragments were found to be adaptably substitutable, and how many of those exploited the full generality of the reference function.

function name	adapter found				no adapter	timeout
	# (full)	#cluster	steps	total time	#	#
tile_pos	6413 (20)	1006	8.9	76.7	24579	15831
dirac_pic_ n_bef_m	1275 (37)	41	6.9	15.8	45473	75
ps_id_ to_tk	894 (0)	9	3.8	4.4	45621	308
leading_ zero_count	946 (0)	16	4.8	5.6	45812	65
trailing_ zero_count	961 (0)	10	4.2	8.6	45782	80
popcnt_32	894 (0)	13	4	7.1	45909	20
parity	894 (81)	14	4	7.1	45911	18
dv_audio_ 12_to_16	894 (0)	9	3.4	6.9	45803	126
is_power_2	894 (33)	8	4.1	8.4	45886	43
Render RGB	1661 (2)	110	6.3	12.0	45138	24
decode_ BCD	894 (0)	12	4.2	8.8	45900	29
mpga_get_ frame_samples	902 (0)	12	3.6	8.2	45718	203

Table 2.7: Reverse engineering using arithmetic adapter substitution for constructing the reference function arguments: This experiment used 46831 target code fragments from a Rockbox firmware image and the first 12 of 24 reference functions from VLC media player grouped by the three overall possible terminations of adapter synthesis. The *#(full)* column reports how many code fragments were found to be adaptably substitutable, and how many of those exploited the complete functionality of the reference function.

function name	adapter found				no adapter	timeout
	# (full)	#cluster	steps	total time	#	#
clamp	5450 (227)	592	9.4	215.0	22890	18483
prev_ pow_2	719 (0)	11	4.4	44.8	35793	18
abs_diff	1012 (5)	31	3.5	22.2	45789	22
bswap32	968 (10)	26	3.3	25.8	45837	18
integer_ cmp	924 (0)	17	3.2	32.8	42234	3665
even	1205 (21)	25	4	39.0	45595	23
div255	880 (0)	12	3.3	26.9	45926	17
reverse_ bits	1190 (0)	13	4.1	31.9	45614	19
binary_ log	905 (0)	13	3.5	34.6	7724	38194
median	3661 (16)	345	6.6	141.7	11300	31862
hex_ value	880 (0)	8	3.3	28.9	39564	6379
get_descr_ len_24b	906 (0)	15	3.8	27.1	45544	373

Table 2.8: Reverse engineering using arithmetic adapter substitution for constructing the reference function arguments: This experiment used 46831 target code fragments from a Rockbox firmware image and the second 12 of 24 reference functions from VLC media player grouped by the three overall possible terminations of adapter synthesis. The *#(full)* column reports how many code fragments were found to be adaptably substitutable, and how many of those exploited the complete functionality of the reference function.

function name	adapter found				no adapter	timeout
	# (full)	#cluster	steps	total time	#	#
tile_pos	4926 (5)	1100	7.6	191.6	9971	31926
dirac_pic_ n_bef_m	1174 (2)	17	4.8	38.2	45551	98
ps_id_ to_tk	917 (0)	14	4	36.1	45289	617
leading_ zero_count	924 (4)	12	4.3	33.2	45517	382
trailing_ zero_count	881 (0)	12	3.9	50.3	6220	39698
popcnt_32	916 (0)	13	3.5	29.3	45890	17
parity	880 (31)	8	3.4	29.6	45923	20
dv_audio_ 12_to_16	880 (0)	7	3	29.6	43426	2517
is_power_2	880 (33)	10	3	28.2	45466	477
Render RGB	5852 (10)	531	5.3	34.5	40948	23
decode_ BCD	920 (0)	35	3.8	33.8	45886	17
mpga_get_ frame_samples	878 (0)	9	3.1	31.1	44453	1348

The full set of results for the type conversion adapter with return value substitution is presented in Tables A.1 and A.2, A.3 and A.4, A.5 and A.6 for the "adapter found", "insubstitutable", and "timeout" conclusions of the adapter synthesis algorithm respectively. The analogous set of results for the arithmetic adapter with return value substitution is presented in Tables A.7 and A.8, A.9 and A.10, A.11 and A.12 in Section A.0.2 of the Appendix.

We present a summary of our results using the type conversion with return value substitution adapter in Tables 2.5, 2.6. We present a similar summary of our results using the arithmetic with return value substitution adapter in Tables 2.7, 2.8. The first column shows the reference functions chosen from the VLC media player source code. The *#(full)* column reports how many code fragments were found to be adaptably substitutable (represented by the value for #), and how many of those exploited the complete functionality of the reference function (represented by the value of *full*). We report average number of steps and average total running time in the *steps* and *total time* columns respectively.

Clustering using random tests

For every target code fragment and reference function pair, we can either find an adapter, find the fragment to not be adaptably substitutable, or run out of time. Our adapter synthesis tool found adaptable substitution using 18 out of the 24 reference functions. For every reference function, we clustered its adapted versions using 100,000 random tests. All adapted versions of a reference function that report the same output for all inputs were placed in the same cluster. The number of clusters is reported in the *#clusters* column. For each reference function, we then manually examined these clusters to judge which adapted versions used the complete functionality of that reference function; these are the cases where describing the functionality of the target fragment in terms of the reference function is mostly likely to be concise and helpful. This took us less than a minute of manual effort for each reference function because we understood the intended semantic functionality of every reference function (we had its source code). We found instances of adapters using the complete functionality of the reference function for 12 reference functions when using the type conversion adapter and for 11 reference functions when using the arithmetic adapter. Reference functions for which we found

no use of complete functionality are omitted in Tables 2.5, 2.6 and 2.7, 2.8.

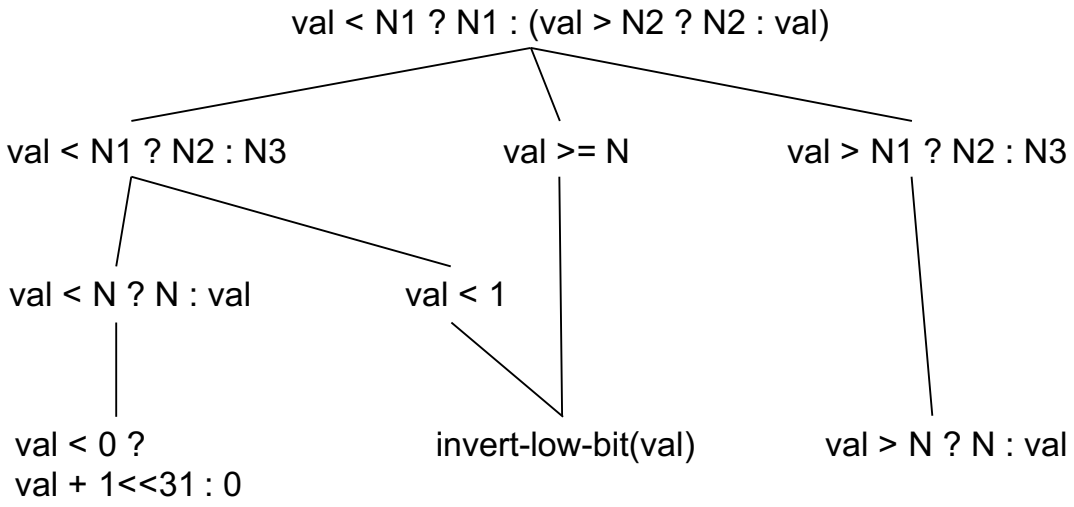


Figure 2.9: Subset of partial order relationship among adapted clamp instances

We found that a majority of our generated adapters exploit specific functionality of the reference functions. We explored this observation further by manually summarizing the semantics of the 683 adapters reported for `clamp`. We found that these 683 adapters have a partial order between them created by our adapter families of type conversion and return value substitution. We present a subset of this partial order as a Hasse diagram in Figure 2.9 with the most general implementation of `clamp` as the topmost node and functions that use the most specific instances of `clamp` at the bottom. To explain one unintuitive example, the `invert-low-bit` operation on a value `v` can be implemented in terms of `val < N` by setting `val` to the low bit of `v` zero-extended to 32 bits and `N` to 1, and zero-extending the low 1 bit of the return value of `val < N` to 32 bits. Some such functionalities owe more to the flexibility of the adapter family than they do to the reference function. These results suggest it would be worthwhile in the future to prune them earlier by searching for instances of the simplest reference functions first, and then excluding these from future searches.

While both the arithmetic and type conversion adapters also allow argument substitution, the arithmetic adapter allows other operations that can be represented as type conversions. However, the arithmetic adapter family does not cover the space of all the

type conversions possible under the type conversion adapter family. We observed one such unexpected occurence with the arithmetic adaptation of some fragments using the `abs_diff` function. This function computes an absolute value of the difference between two arguments. We found that the arithmetic adapter can pass the difference of two arguments as the first argument and the constant `0` as the second argument to the `abs_diff` function to adaptably substitute some code fragments using the `abs_diff` reference function. We show an example of such an adapted `abs_diff` function in Figure 2.10. Figure 2.10 shows the source code of an arithmetic adapter wrapping the `abs_diff` function inlined in the disassembly of the adapter's assembly code. Lines 8-11 load the two arguments to be passed to `abs_diff` from the stack and subtract the second argument from the first one, placing the result into `r0` on line 11. Line 12 loads the constant value 0 into the second argument register (`r1`). Line 13 calls the `abs_diff` function using the adapted arguments. Lines 14 and 16 place the return value of `abs_diff` into the return value register of `adapted_abs_diff` and return the control-flow to the caller.

```
1  int adapted_abs_diff(int a, int b) {
2     105f8:       e92d4800        push    {fp, lr}
3     105fc:       e28db004        add     fp, sp, #4
4     10600:       e24dd008        sub     sp, sp, #8
5     10604:       e50b0008        str     r0, [fp, #-8]
6     10608:       e50b100c        str     r1, [fp, #-12]
7    return abs_diff(a-b, 0+0);
8     1060c:       e51b2008        ldr     r2, [fp, #-8]
9     10610:       e51b300c        ldr     r3, [fp, #-12]
10    10614:       e0633002        rsb     r3, r3, r2
11    10618:       e1a00003        mov     r0, r3
12    1061c:       e3a01000        mov     r1, #0
13    10620:       ebffffe6        bl      105c0 <abs_diff>
14    10624:       e1a03000        mov     r3, r0
15 }
16    10628:       e1a00003        mov     r0, r3
17    1062c:       e24bd004        sub     sp, fp, #4
18    10630:       e8bd8800        pop     {fp, pc}
```

Figure 2.10: The `abs_diff` function adapted using an arithmetic adapter.

Timeouts were the third possible conclusion of each adapter synthesis task. The number of timeouts is reported in the "timeouts" column in Tables 2.5, 2.6 and 2.7, 2.8. We show a histogram of the total running time of each adapter synthesis task, that ended

with finding an adapter using the type conversion adapter with return value substitution, in Figure 2.11 for the `clamp` reference function. Similar histograms for `tile_pos` and `median` reference functions can be found in Figures A.1 and A.2 in Section A.0.3 in the Appendix. The arithmetic adapter also had a large number of timeouts with the `clamp`, `tile_pos`, and `median` functions. We present similar histograms when an arithmetic adapter was used for the `clamp`, `tile_pos`, `median`, and `trailing_zero_count` reference functions in Figures A.3, A.4, A.5, and A.6 respectively in Section A.0.3 in the Appendix. We attribute this phenomenon to the arithmetic adapter being more difficult for the solver to synthesize as compared to type conversion. This hypothesis is confirmed by data presented in column "AS total time" in Tables A.11 and A.12 in Section A.0.2 in the Appendix. The adapter synthesis for `median` spent 537.6 of its allocated 600 seconds in solving queries during adapter search steps on an average. Similarly, the adapter synthesis for `tile_pos` spent 549.3 of its allocated 600 seconds in solving queries during adapter search steps on an average.

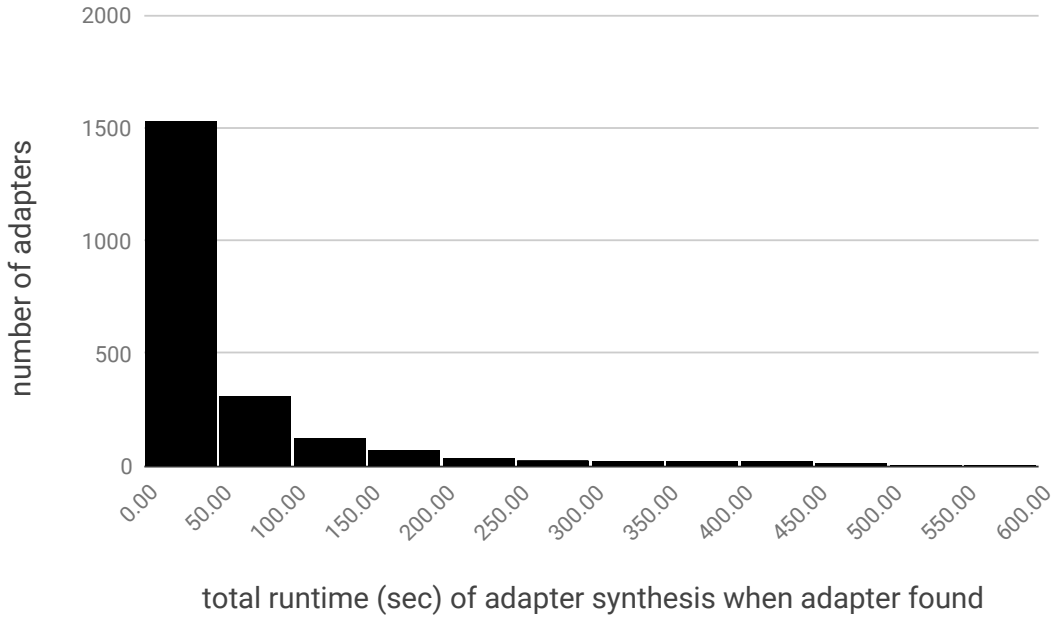


Figure 2.11: Running times for synthesized adapters using `clamp` reference function. This figure shows that a majority of adapters are synthesized within the first 300 seconds of the total time budget of 600 seconds.

Figure 2.11 shows that the number of adapters found after 300 seconds decreases rapidly, consistent with the mean total running time (subcolumn *total time* under column *adapter* in Tables 2.5, 2.6) of 99.3 seconds for the `clamp` reference function. Tables 2.5, 2.6 also show that the total running time, when our tool concludes with finding an adapter, is significantly less than 600 seconds for all reference functions that reported adapters. Though setting any finite timeout can cause some instances to be lost, these results suggest that a 600-second timeout was appropriate for this experiment, and that most timeouts would not have led to adapters.

2.4.8 Comparing adapter families

adapter family	size	#adapters	#inequiv.	#timeout
arg_perm+ ret_sub-2.5m	4.98E+10	9	46803	19
arg_sub+ ret_sub-2.5m	1.3538427E+126	705	45782	344
type_conv+ ret_sub-5m	1.3538430E+126	683	40553	5595

Table 2.9: Comparing adapter families with 46,831 target code fragments and `clamp` reference function

We also explored the tradeoff between adapter search space size and effectiveness of the adapter family. We ran all 46,831 target code fragments with `clamp` as the reference function using two additional adapter families beyond the combination of type conversion family with return value substitution described above. The first adapter family allowed only argument permutation and the second allowed argument permututation along with substitution with unrestricted 32-bit constants. We ran the first adapter family setup (argument permutation + return value substitution) with a 2.5 minute hard time limit, the second adapter family setup (argument substitution + return value substitution) with a 5 minute hard time limit, and the third adapter family setup (argument substitution + return value substitution) was the same as the previous subsection with also a 5 minute hard time limit. We present our results in Table 2.9. As expected, the number of timeouts increases with an increase in the size of adapter search space. Table 2.9 also shows that, for `clamp`, a simpler adapter family is better at finding adapters than a more expressive family, because more searches can complete within the timeout. But, this may not be true for all reference functions. Table 2.9 suggests that, when computationally feasible, adapter families should be tried in increasing order of expressiveness to have the fewest timeouts overall. We plan to explore this tradeoff between expressiveness and effectiveness of adapter families in the future.

2.5 Discussion

To check equivalence of side-effects of the pair of functions on the operating system, our adapter synthesis tool checked for equality of a sequence of system call numbers. Our equivalence checking can be extended to look for semantic, instead of exact, equivalence of system calls. In some cases, different system calls may be equivalent with certain parameters. For example, the *creat* system call is equivalent to the *open* system call with its second argument set to *O_CREAT—O_WRONLY—O_TRUNC*. Allowing for more relaxed notions of equivalence between system calls is non-trivial because of the many diverse side-effects that system calls can have on the filesystem and operating environment. We can implement such side-effect equivalence checking by adding more thorough emulation of system call execution to FuzzBALL.

Functions which use the same system calls can impose different preconditions on the system call arguments. e.g. we found that while the *killpg* C library function internally calls the *kill* function, it does this only if its first process group argument is greater than or equal to 0. Adding knowledge of such preconditions to our adapter search can make our adapter synthesis implementation more robust but lies outside the scope of adapter synthesis. While the presence of imposed pre-conditions can be detected from the implementation of a function, this problem becomes more difficult when functions use different values to represent the same semantic notion. e.g. While *isalpha* and *isalnum* will return a non-zero value for an alphabetic character passed as an argument, they return different non-zero values to represent a boolean value *true*. More expressive adapter grammars are required to be applied on the return value to allow synthesis of such post-conditions.

2.5.1 Interpreting adapter synthesis conclusions

A negative result is said to occur when adapter synthesis fails to synthesize an adapter, given an adapter family. However, we cannot label this as a true negative or false negative because the conclusion of adapter synthesis depends on the adapter family used to run it. It may still be possible for that target function to be adaptably substituted with the same reference function using a adapter family more sophisticated than is currently implemented by us.

On the other hand, a positive result is said to occur when adapter synthesis finds an adapter that allows an adapted reference function to substitute a target function or fragment. However, a positive result cannot be labeled as being a true positive or false positive either. The definition of a true or false positive depends on whether or not knowledge of an adapted reference function is useful to the reverse engineer. We used the assumption that a reverse engineer would only be interested at looking at adapters that use the complete functionality of the reference function. We reduced the number of adapted reference functions that should be reported to a reverse engineer by clustering adapted reference functions using random tests. The clustering reduced human reverse engineering effort because only one adapted reference function from each cluster had to be manually examined. However, this assumption is subjective to us and was not validated by human experts. We did not validate our results by running them through a panel of human reverse engineering experts.

2.5.2 Improving memory substitution performance

Our implementation of the memory substitution adapter relies on being told which arguments to the target function are pointers. It then uses concrete addresses for pointer arguments and keeps track of counterexamples in memory. In order to resolve this limitation, we need to implement a pre-processing step that classifies target function arguments as being pointers and passes this information into memory adapter synthesis.

Another limitation, which is a crucial optimization used by our memory substitution adapter implementation, is being told which of the target function's pointer arguments points to a structure that needs to be adapted. This limitation is less fundamental because we can simply tell the adapter synthesis to synthesize adapters for all memory pointed to by the target function's pointer arguments. We would then expect identity memory substitution adapters for non-structure memory pointed to by pointer arguments to the target function when adapter synthesis succeeds for a target/reference function pair. However, it would make the overall memory substitution adapter synthesis slower because of having to unnecessarily synthesize identity memory substitution adapters.

A third limitation is that our memory substitution adapter synthesis is parameterized by the maximum size of the structure between the target and reference functions

and the number of fields. The larger these parameters, the more expensive the adapter synthesis process. In the future, we plan to integrate a preprocessing step that infers a tight bound on the size of structure arguments and a tight bound on the number of fields in each structure argument. There are number of other improvements we plan to make to optimize the performance of our memory substitution adapter synthesis.

On combining the memory substitution adapter family with argument and return value substitution, our adapter synthesis tool encountered a significant slowdown with both RC4 context initialization and encryption. This can be attributed in part to the encoding of memory substitutions in our tool. We enumerate all possibilities of memory substitutions into the formula of every byte in the memory substitution structure, causing the symbolic formulas to be very large. We plan to encode the memory substitution adapter more efficiently in the future to make better use of existing solvers. Another significant cause of the slowdown, in the case of RC4 context initialization, is the slow symbolic execution of 256 rounds of key mixing, once in the target and once in the reference function, because of two symbolic loads and two symbolic stores to memory on every round of key mixing. We plan to integrate loop summarization (for example, as described by Godefroid et al. [48]), and use the theory of arrays for symbolically-indexed memory accesses to speed up this symbolic execution in the future. In the case of RC4 encryption, since we have to adapt the memory substitution structures twice (once for input and once for output) we have to present large formulas to the solver at least twice on every execution path with each query taking a few seconds. This large symbolic state is the cause of significant slowdown during RC4 encryption adapter synthesis. We plan to explore concretization heuristics of symbolic bytes in the future to reduce the number of solver invocations made during RC4 encryption adapter synthesis. Finally, the memory substitution adapter performance is currently optimized by having the user specify a tight upper bound for the structure and the number of fields in the structure. In the future, we plan to alleviate this limitation by adding a pre-processing step that infers these bounds and specifies them to the memory substitution adapter search.

2.5.3 Translating adapters to binary code

We currently represent our synthesized adapters by an assignment of concrete values to symbolic variables and manually check them for correctness. Adapters could instead be

automatically incorporated into binary code to replace the original function with the adapted function. This would make the adapters more convenient to use and easier to test automatically. We plan to automate generation of such adapter code in the future.

2.5.4 Applications of adapter synthesis

Since adapter synthesis introduces flexibility when searching for a substitution, it can be extended for more applications. Binary program repair is a promising application as shown by our case study for using adapter synthesis to perform substitution modulo a bug. Adapter synthesis allows the user to find reference binary code that provides the same functionality as the target code but does not have the buggy undesirable behavior. Being able to repair security bugs in binary code can potentially allow users of commercial software to repair bugs in their purchased software by using other binary code as reference. Binary program repair also has the potential to allow users of commercial software to protect themselves from security bugs by using a more complete adapter synthesis tool that can find reference binary code that does not have any bugs and also patch it in the commercial software. Another promising direction is using adapter synthesis for deobfuscation. By using a obfuscated binary code fragment as the target, if adapter synthesis can find a function with known source code to be the adapted reference substitution, then it can effectively allow the reverse engineer to deobfuscate the binary code fragment. Finally, adapter synthesis can also be applied at source-level for library replacement. The availability of type information can make it easier to synthesize more expressive adapters and allow software developers to migrate library dependencies automatically.

2.5.5 Improving adapter synthesis performance

The depth of our arithmetic adapters was limited to using a single unary or binary operator. This limitation could have caused target/reference pairs to be reported as being inequivalent when they could have been adapted by a more expressive arithmetic adapter. In the future, we plan to use a less expressive arithmetic adapter family to filter out target code that can be adapted by it. Only target code that is reported as being inequivalent by the less expressive arithmetic adapter family then needs to be

tried for adaptable substitution with more expressive arithmetic adapter families. Our evaluations also showed that using more expressive adapters takes longer to synthesize. In the future, we plan to address this limitation by constructing an ordering of adapter families that ranges from less expressive to more expressive. We would then use this ordering to attempt synthesize more expressive adapters when synthesis of less expressive adapters has failed.

During every adapter search step, symbolic execution explores all feasible paths, including paths terminated on a previous adapter search step because they did not lead to a correct adapter. Once a candidate adapter is found, the next iteration of adapter search can be accelerated by using information saved from the previous iteration. For example, adapter search can pick up symbolic execution from the last path in the previous iteration that led to a correct adapter. A similar optimization can be utilized for concrete enumeration-based adapter search that uses the same random order of adapters during an adapter synthesis run. Concrete enumeration-based adapter search can be further accelerated by searching for adapters in parallel since checking one adapter is independent of checking other adapters.

We can also easily extend our adapter grammar to allow operations on floating point values. This would allow us to synthesize adapters between functions that implement mathematical formulas and between function pairs such as *frexpf* and *frexp*. Adding type conversion operations to floating point adapter grammars would allow us to find argument substition adapters between function pairs such as *frexpf* and *frexp*.

Our tool currently assumes that all behaviors of the target function must be matched, modulo failures such as null dereferences. Using a tool like Daikon [49] to infer the preconditions of a function from its uses could help our tool find adapters that are only correct for correct uses of functions. This would allow us to find equivalence between functions like *isupper* and *islower*, which expect certain types of input.

Adapter synthesis requires us to find if *there exists* an adapter such that *for all* inputs to the target function, the output of the target function and the output of the adapted reference function are equal. Thus the synthesis problem can be posed as a single query whose variables have this pattern of quantification (whereas CEGIS uses only quantifier-free queries). We plan to explore using solvers such as Yices [50] for this $\exists\forall$ fragment of first-order bitvector logic.

Symbolic execution can only check equivalence over inputs of bounded size, though improvements such as path merging [4, 5] can improve scaling. Our approach could also integrate with any other equivalence checking approach that produces counterexamples, including ones that synthesize inductive invariants to cover unbounded inputs [51], though we are not aware of any existing binary-level implementations that would be suitable.

2.6 Related Work

2.6.1 Detecting Equivalent Code

The majority of previous work in this area has focused on detecting *syntactically* equivalent code, or 'clones,' which are, for instance, the result of copy-and-paste [52–54]. Applications of detecting functionally equivalent code (aside from our motivating application of multivariant execution) include functionality-based refactoring, semantic-aware code search, and checking 'yesterday's code against today's.' Jiang et al. [55] propose an algorithm for automatically detecting functionally equivalent code fragments using random testing and allow for limited types of adapter functions over code inputs — specifically permutations and combinations of multiple inputs into a single struct. Similar to our work, both [55] and [26] define functional equivalence based on input and output behavior. The key difference between our approach and [55] is that we rely on symbolic execution as opposed to random testing, and that we allow for more interesting adapter functions over code inputs. Ramos et al. [26] present a tool that checks for equivalence between arbitrary C functions using symbolic execution. While our definition of functional equivalence is similar to that used by Jiang et al. and Ramos et al., our adapter families capture a larger set of allowed transformations during adapter synthesis than both.

Amidon et al. [56] describe a technique for fracturing a program into pieces which can be replaced by more optimized code from multiple applications. They mention the need for automatic generation of adapters which enable replacement of pieces of code which are not immediately compatible. While Amidon et al. describe a parameter reordering adapter, they do not mention how automation of synthesis of such adapters

can be achieved. David et al. [57] decompose binary code into smaller pieces, find semantic similarity between pieces, and use statistical reasoning to compose similarity between procedures. Since this approach relies on pieces of binary code, they cannot examine binary code pieces that make function calls and check for semantic similarity across wrappers around function calls. Goffi et al. [58] synthesize a sequence of functions that are equivalent to another function w.r.t a set of execution scenarios. Their implementation is similar to our concrete enumeration-based adapter search which produces equivalence w.r.t. a set of tests. In the hardware domain, adapter synthesis has been applied to low-level combinatorial circuits by Gascón et al [59]. They apply equivalence checking to functional descriptions of a low-level combinatorial circuit and reference implementations while synthesizing a correct mapping of the input and output signals and setting of control signals. They convert this mapping problem into a exists/forall problem which is solved using the Yices SMT solver [50]. More recently, Katz et al. [60] have applied machine learning to the problem of decompilation of binary code. Their technique predicts decompiled source code, given a fragment of binary code. A primary difference between adapter synthesis and the technique presented by Katz et al. is that, if substitutability is found by adapter synthesis, the match will be exact, whereas the Katz et al's technique finds an approximate match which may not be usable for applications such as library replacement. Their permutation of input signals and output signals are similar to our argument substitution and return value adapters. However, their technique depends on the user specifying input and control signals for reference implementations whereas our technique does not depend on any such prior classification.

2.6.2 Variant Generation

Another similar approach to developing variants relies on compiler-based randomization [61]. It can be convenient to modify a compiler to support randomization because compilers already have support for many of the analyses required for randomization and are set up 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. Because we check for functional equivalence at the binary level, our approach does not require source code and is compatible with proprietary compilers. Compiler-based approaches to diversity are also limited in

the types of diversity they can introduce.

2.6.3 Component Retrieval

Component retrieval is a technique [62], [63], [64] that provides a search operator for finding a function, whose polymorphic type is known to the programmer, within a library of software components. The search results contain components whose types are similar but more general (or more specialized). Adapter synthesis shares the same intuition in that, it adapts the more general implementation of a functionlity to the more specific one. Type-based hot swapping [65] and signature matching [66] are related areas of research that rely on adapter-like operations such as currying or uncurrying functions, reordering tuples, and type conversion. Reordering, insertion, deletion, and type conversion are only some of the many operations supported by our adapters. These techniques can only be applied at the source level, whereas our adapter synthesis technique can be applied at source and binary levels The earliest related work on type-based component retrieval was by Rittri [62]. Given a query type A, Rittri presents a technique for searching through a library and finding all identifiers whose type is isomorphic to A. This is done by modeling the problem as a Cartesian Closed Category (CCC) that has the product, exponentiation, and terminal objects defined and using products in normal forms as bags (multisets) which can then be compared for equality. This paper actually mentions the need to have the search system provide a conversion function to convert the library value into the type of the user's query. This paper mentions work done by Runciman et al. [63] which allows the search system to include a search result where the library function has an extra argument which corresponds to a value that is constant in the user's application. These rules come together to form our argument substitution adapter family. In an extended 1991 paper, Runciman et al. [64] present a technique for finding a function, whose polymorphic type is known to the programmer, within a library of software components. The technique was evaluated at the source level. The presented search operator allows a programmer to explore a library for near-misses, and presents the programmer with a list of search results. The search results contains components whose types are similar but more general (or more specialized). As mentioned by Runciman et al., their technique finds the curried form of a function that is more general, and is complementary to an approach that allows

re-ordering, insertion, and deletion. However, re-ordering, insertion, and deletion are only some of the operations performed by our adapters. adapter synthesis is also related to type-based hot swapping of running modules in long-lived applications. Duggan [65] presents a technique for swapping modules while they are running where the swap must be type-based.This technique verifies the type-based swap to be correct on the basis of the version adapters provided by the user. The version adapters that this technique uses are exactly the same as our type conversion adapters. However, this technique depends on the version adapters being supplied for performing the type-based swap whereas our technique searches for the correct adapter. Signature matching is another problem which bears resemblance to adapter synthesis. Zaremski et al. [66] present a technique for finding functions or components in a library that match the user's query type. While an exact match of a search function type is useful, the technique can find more interesting search results by allowing for partial matches with relaxations such as generalized match, specialized match, and relaxations that allow transformations such as currying or uncurrying functions, reordering tuples. The reordering relaxation is similar to our argument substitution adapter family. Both of our techniques have similar applications, they lead to better program understanding, function reuse. However, our technique performs equivalence checking of the target and adapted reference functions, whereas their technique only matches type information. They mention that their technique can be combined with a specification matcher to search for specification matching only between function pairs that were returned as valid search results by their tool. Also, our technique is implemented at the binary level, whereas their technique is more likely to be useful as-is at the source level.

2.6.4 Component Adaptation

Component adaptation is another related area of research, that given a formal specification of a query component, searches a library of components within a set of adaptation architecture theories. This includes techniques for adapter specification [67], for component adaptation using formal specifications of components [68], [69], [70], [71], [72]. Component adaptation has also been performed at the Java bytecode level [73], as well as at low-level C code [74]. Behavior sampling [75] is a similar area of research for finding equivalence over a small set of input samples. However, these techniques either

relied on having a formal specification of the behavior of all components in the library to be searched, or provided techniques for translating a formally specified adapter [67].

One of the early works on component adaptation was by Purtilo et al. [67]. They presented a language called Nimble which can be used by programmers to specify an interface adaptation for reuse of a module. The module reuse can be done in the same language or across different languages using module interconnection languages. This paper mentions the possibility of using an adapter function which performs the interface adaptation. The interface map is specified by the programmer and the code for the adapter function is generated by Nimble. This paper also mentions that if the bijection from the actual parameters passed by the calling function to the formal parameters taken by the reused function is order-preserving, such that the ith actual argument maps onto the ith formal argument, then the mapping is an isomorphism, and the parameter lists are syntactically equivalent. This is the definition used by the identity adapter which is the default adapter we start searching with. The most significant difference between our work and Nimble is that Nimble requires the programmer to specify the adapter where we search for the adapter in an automated manner. If Nimble generated binary-level adapters, the adapters that our search tool finds could be encoded in the Nimble language and translated into binary code.

A similar method of component adaptation was described by Morel et al. [68]. Given a formal specification of a component to be searched for and a library of components, SPARTACAS can automatically find an adaptation that reuses components in the library to implement the query component. SPARTACAS implements the adapter search using the sequential, alternative, and parallel adaptation architecture theories as described by Penix et al [69]. Our adapter search technique can be extended further by implementing these adaptation architecture theories. For example, the sequential adaptation architecture can be used by passing the return value of one reference function as an argument to a second reference function. The work done by Penix et al. [69] is similar to adapter synthesis. Penix et al. present a technique for automatic component adaptation using specification matching. The adaptation phase makes use of the results of a retrieval mechanism [70] that classifies software components using semantic features derived from their formal specifications. The adaptation phase then searches for a way to adapt or combine components returned by the retrieval mechanism to solve the formal

specification of a problem. Formal approaches to component adaptation have also been proposed as presented by Yellin et al. [71] and extended by Bracciali et al. [72] These approaches adapt mismatched behavior between two components. Bracciali et al. provide a high-level notation for expressing adapter specifications, and an automated procedure for converting the adapter specifications to concrete adapters. Their work bears similarity to ours because they also express the adapter as a component-in-the-middle, similar to our intuition of the adapter being a wrapper around the reference function. Their one-to-one, many-to-one correspondence and use of the keyword none to indicate asymmetry between components when adapting components, were similar to the operations performed by our argument substitution adapter, when adapting functions. Finally, both our technique and theirs perform adaptation at the interface level. However, our technique adapts the reference function to the target function, whereas their technique adapted mismatched behavior between both components simultaneously. Our technique treats the binary-level implementation of the target and reference functions as their own behavior specification whereas their technique requires the behavior of both components to be formally specified. Our adaptation technique does not depend on the availability of formal specifications of functions and, therefore, is an improvement over the work presented in these component adaptation techniques.

Component adaptation has also been done at the Java bytecode level [73]. In this work, the programmer writes a delta (adapter) specification, which is translated into a modified class bytecode form by a Delta compiler. The allowed operations are renaming of class, methods, fields, changing the super class, add interface to implements clause, add method, add field to the class, and finally, rename a symbolic reference to a field. Java byte code contains all references to classes, interfaces, and fields, type information of all methods, method names, to ensure safe execution of programs. The adaptation is done only if the previously-specified preconditions are met. Thus, adapter search is not automated, but manually defined in an earlier step (adapter specification). Adaptation of data layout has also been done at the bit-level for C code by Nita et al [74]. However, this work also allows a programmer to specify multiple alternative data layouts in a declarative language, instead of automating this search.

Behavior sampling [75] also bears a similarity with adapter synthesis. This technique includes our intuition that the order of parameters is not important as long as there

is one-to-one correspondence between the parameters of the target interface and the formal parameters of the reference interface and the correspondence also respects types and modes. This technique finds behavioral equivalence over a small set of input samples and leaves it up to the user to determine exact equivalence by going through the search results. The user must also define expected outputs for each input in the sample set by computing the output values by hand or some other means. This is in contrast with our technique which finds exact equivalence over all inputs to the target function where the reference function is being adapted to become behaviorally equivalent to the target functions.

2.6.5 Program Synthesis

Program synthesis is an active area of research that has many applications including generating optimal instruction sequences [76, 77], automating repetitive programming, filling in low-level program details after programmer intent has been expressed [10], and even binary diversification [78]. Programs can be synthesized from formal specifications [79], simpler (likely less efficient) programs that have the desired behavior [10,76,77], or input/output oracles [9]. We take the second approach to specification, treating existing functions as specifications when synthesizing adapter functions.

The approach that comes closes to adapter synthesis is the one described by Jha et al. [9]. Given an I/O oracle that can provide the correct output given inputs, Jha et al. attempt to synthesize a program that fits the specification provided by means of I/O examples from the oracle. Jha et al. make use of location variables as part of their process for synthesizing candidate programs. Location variables map to line numbers which are either a line in the synthesized program or some input to the entire component. Adapter synthesis makes use of similar adapter variables which map to inputs and outputs to the reference function.

The general area of program synthesis that adapter synthesis bears most similarity to is syntax-guided synthesis [3]. Program synthesis attempts to synthesize an implementation that matches a given correctness specification. while such synthesis can be done within a reasonable amount of computation time for small programs, it quickly becomes intractable when applied to large-scale programs. Syntax-guided synthesis attempts to use a syntactic constraint in addition to the correctness specification to make

the search for a candidate more tractable. Adapter synthesis borrows the idea of using a syntactic template for an adapter to constrain the space of adapters to be searched. However, adapter synthesis differs from syntax-guided synthesis in two ways.

1. Adapter synthesis makes use of a target program as the specification. This makes it easier to construct the expected output for a given counterexample. Adapter synthesis can simply run the target program to construct the expected output. Finding the expected output for a counterexample is not straightforward for syntax-guided synthesis techniques that rely on only a correctness specification. This presents a minor difference because it is an idea that's used in some syntax-guided synthesis tools too. For example, Alchemist [12] uses a teacher, (similar to the counterexample search used in adapter synthesis) that knows the target concept and can therefore provide a counterexample that discards many candidates from the synthesis search space.

2. Adapter synthesis wraps around a reference implementation to match the specification provided by the target program. This allows adapter synthesis to find adapted reference implementations to match much more complex target specifications than those handled by syntax-guided synthesis in a reasonable time budget.

2.6.6 N-version Systems

The earliest work related to creating function equivalence happened during the 1970s when fault-tolerant computing research began to gain momentum and the idea of n-version programming was proposed [80], [81]. N-version programming is defined as the process of independently generating functionally equivalent programs from the same initial specification. Since then, a variety of work has analyzed the assumption of the independence of errors among different versions [82], the effectiveness of N-version programming in the presence of overlapping errors [83], and the practicality from a software engineering standpoint of developing multiple less-reliable versions versus a single highly-reliable version [84]. Generally, this work finds that running several 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 versions of a program is a good way to achieve fault tolerance 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 [85] to N-version programming have taken advantage of *natural diversity* seen in related software programs. Natural diversity refers to diverse implementations of the same functionality that emerge naturally, typically due to 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. Studies have shown that the naturally-emerging diverse implementations of different commercial-off-the-shelf products are good candidates for N-version programming because they typically have independent bugs [86,87]. We hope to take advantage of natural diversity within a codebase when looking for functionally 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 [88,89]. The distinction between a 'version' and a 'variant' is that a version is manually developed while a variant is automatically generated. Cox et al. [88] show how to construct variants, so that they have disjoint vulnerabilities with respect to certain classes of attacks, using two randomization 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. [89] 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 stack growths and system call number randomization. One restriction to the MVEE is that it requires all variants to make the same system calls, in the same order, with the same arguments.

Automated variant construction avoids the overhead of manual development of multiple program versions and enables more formal arguments about a system's security, but 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 do not introduce diversity at the

design level, i.e. at the level of data structures and algorithms. We believe that design-level diversity is also an important source of protection against attacks and we hope to discover design-level diversity through our search for equivalent functions. However, to avoid the cost of manually developing different program implementations, we want to use the diverse equivalent functions we find in an automated way.

2.7 Conclusion

We presented a new technique to search for semantically-equivalent pieces of code which can be substituted while adapting differences in their interfaces. This approach is implemented at the binary level, thereby enabling wider applications and consideration of exact run-time behavior. We implemented adapter synthesis for x86-64 and ARM binary code. We presented examples demonstrating applications towards adaptation modulo a bug, library replacement, and reverse engineering. We present an evaluation to find substitutable code within a library using the C library. Our adapter families can be combined to find sophisticated adapters as shown by adaptation of RC4 implementations. While finding thousands of functions to not be equivalent, our tool reported many instances of semantic equivalence, including C library functions such as *ffs* and *ffsl*, which have assembly language implementations. Our comparison of concrete enumeration-based adapter search with binary symbolic execution-based adapter search allows users of adapter synthesis to choose between the two approaches based on the size of the adapter search space. Our case studies show that adapter synthesis can be applied at scale to reverse engineer binary code using independently-developed codebases, even in the presence of very large adapter search spaces. Our implementation constitutes a novel use of binary symbolic execution for synthesis. Our results show that the CEGIS approach for adapter synthesis of binary code is feasible and sheds new light on potential applications such as searching for efficient clones, deobfuscation, program understanding, and security through diversity.

Chapter 3

Automatically Summarizing Adapters Using Path-Merging

3.1 Introduction

Adapter synthesis uses the Counterexample-guided Inductive Synthesis (CEGIS) algorithm for synthesizing an adapter. It consists of two deductive procedures. (1) A verification step that checks the last candidate adapter for correctness and reports a counterexample if it is found to be incorrect. (2) An adapter search step that searches for an adapter that allows an exact substitution of the target code with the adapted reference code. In the implementation used for the evaluation in Chapter 2, the adapter formulas were implemented directly into the binary symbolic executor, FuzzBALL. For example, the formulas presented in Listings 2.8 and 2.9 perform argument substitution and type conversion on the target arguments to create arguments to be passed to the reference code. While we found these adapter families to be useful, users of our adapter synthesis tools are restricted to using only the adapter families implemented in FuzzBALL. A counter-argument to our implementation strategy is: why not write the adapter code in a source-level language that can be compiled to binary code. The binary symbolic executor can then simply symbolically execute the adapter as part of its adapter search step. We present an example of such adaptation code performing argument substitution in Java in Figure 3.1.

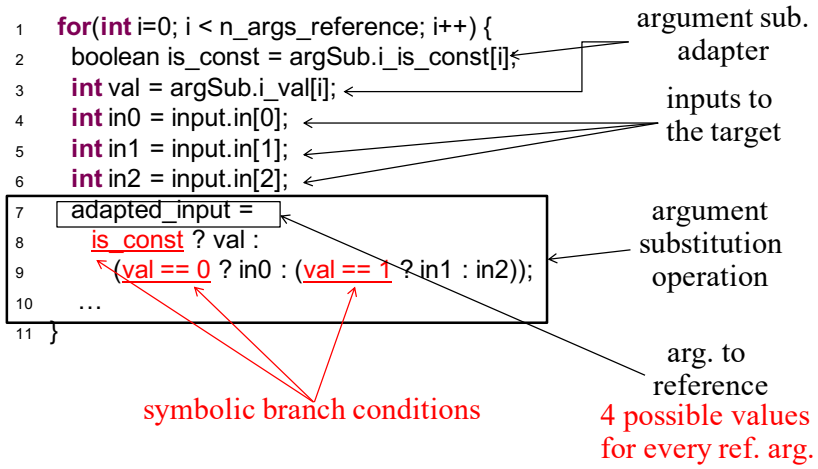


Figure 3.1: An example of argument substitution adaptation in Java

Lines 2 and 3 place the adapter variables, similar to `y_1_type` and `y_1_val` in Listing 2.8, into local variables,`is_const` and `val`. Next, assuming that the target had three inputs, the inputs are placed into local variables, `in0`, `in1`, `in2`. Finally, an argument substitution adaptation is done which chooses one of `in0`, `in1`, and `in2`, or the constant value specified in `val` to set to the adapted input to the reference. During an adapter search operation, the `is_const` and `val` variables will be symbolic because the adapter search attempts to construct an adapted input using the target inputs, `in0`, `in1`, and `in2`. This will cause the underlined branches on lines 8 and 9 to be symbolic branches which will have both sides feasible. For creating every adapted input, symbolic execution will have to explore 4 execution paths, with each path setting the adapted input to one of four possibilities, `val`, `in0`, `in1`, and `in2`. This turns into 4^n execution for n reference arguments.

The implementation in Chapter 2 attempted to avoid this path explosion by encoding the adapter formulas directly into the binary symbolic executor during the adapter search step. However, this approach trades off flexibility for performance. In order to find a middleground between flexibility and performance, it is desirable to automatically summarize the adaptation operation into a single formula when possible. This automatic summarization will avoid path explosion resulting from the adaptation and also allow

users of the adapter synthesis tool to write their own adaptation operation in a source-level language that can, later, be compiled to binary code. We have implemented path-merging in a symbolic executor named *Java Ranger*, the details of which will be presented in Chapter 4. In the following section, we present a case study that allows the user to specify argument substitution in Java, the summary for which can be automatically computed and applied by Java Ranger.

3.2 Automatically summarizing adapter formulas

We implemented adapter synthesis into Java Ranger to evaluate if path-merging allows users to specify an adapter directly at the source-level while avoiding the path explosion from adapter formulas during adapter search. We evaluated our implementation using a pair of functions presented in Listing 3.1.

```
1 public int target_fn(int x, int y) {
2   return (x << 1) + (y&255)%2;
3 }
4 public int ref_fn(int a, int b, int c, int d) {
5   return a + b + (c & d);
6 }
7 public int adapted_ref_fn(int x, int y) {
8   return ref_fn(x, x, y, 1);
9 }
```

Listing 3.1: Pair of Java functions used to evaluate the use of path-merging during adapter search

Our implementation used Java Ranger for both the counterexample search and adapter search steps shown in Figure 2.1. Java Ranger can find the right adapter shown in lines 7–9 of Listing 3.1 with 4 invocations of couterexample search and 3 invocations of adapter search. We report our results in Table 3.1.

	A.S.#1 (time (s), # exec. paths)	A.S.#2 (time (s), # exec. paths)	A.S.#3 (time (s), # exec. paths)	Total (time (s), # exec. paths)	Average (time (s), # exec. paths)
Adapter Synthesis w.o. path-merging	(3, 86)	(4, 168)	(8, 492)	(15, 746)	(8.3, 490.4)
Adapter Synthesis with path-merging	(9, 143)	(9, 144)	(9, 152)	(27, 439)	(10.4, 152.4)

Table 3.1: Adapter search time and number of execution paths for the three adapter search steps required to find the right adapter for the pair of (target, reference) functions shown in Listing 3.1.

Table 3.1 shows that during the first adapter search step takes fewer execution paths to find the right adapter without path-merging than with path-merging. However, this relation is flipped as adapter search is performed for more than one test. In terms of time, every adapter search step involves a static analysis step that produces an overhead of about 4 seconds of running time, the cost of which gets amortized over adapter search steps over more test inputs. This is shown by the third adapter search step which takes almost the same time with path-merging as the time needed to find an adapter without path-merging. In order to the influence of the symbolic executor without path-merging getting "lucky" with finding the right adapter, we also performed the same adapter synthesis 10 times with and without path-merging. We report these results in the "Average (time (s), # exec. paths)" column in Table 3.1. Table 3.1 shows that the difference between the average running time with path-merging and with path-merging is about 2 seconds but path-merging reduces the number of execution paths by about 69%. With this significant reduction in number of execution paths, path-merging

successfully avoids the path explosion seen from constructing adapted arguments while still allowing the adapter to be specified in a source-level language such as Java.

3.2.1 Accelerating adapter synthesis-based reverse engineering with path-merging

In chapter 2.2 Section 2.4.7, we showed adapter synthesis was useful in reverse engineering large fragments of binary code taken from a firmware image built for the iPod Nano 2g device. This evaluation used reference functions from the source code of VLC media player. While adapter synthesis was able to find adaptable substitutability for many of the target fragments, it also ran into several instances of timeouts with a 5 minute time limit. The most number of timeouts happened with the `BinaryLog`, `median`, and `trailing_zero_count` reference functions. We randomly selected 5 target fragments with which adapter synthesis timed out with these 3 reference functions when synthesizing the arithmetic adapter. This process gave us 15 pairs of the form: (target fragment, reference function). For each of these pairs, we re-ran the adapter synthesis, this time giving it a 24-hour time limit. 11 of the 15 pairs completed adapter synthesis with the "insubstitutable" conclusion within 24 hours but 4 of them still timed out. All 4 of these adapter synthesis tasks were attempting to synthesize an adapter using the `median` reference function. We wanted to evaluate if path-merging can accelerate the process of not just finding an adapter, but also concluding that no adapter exists, since this conclusion also involves running an adapter search step to completion. Since our implementation of adapter search uses FuzzBALL to find an adapter and FuzzBALL does not have the path-merging feature, we modified the implementation of `median` to make its compiled binary code reflect the same benefits as path-merging. Specifically, when compiled, the binary code of our hand-modified `median` did not use any branching to compute the median of three values. We ran the 4 adapter synthesis tasks that had timed out even with a 24-hour time limit using our hand-modified non-branching implementation of `median` as the reference function. We present the results of this evaluation in Table 3.2.

	target frag #1	target frag #2	target frag #3	target frag #4
total time (s)	2724	816	2242	1941
last AS time (s)	1003	267	513	1035

Table 3.2: Results of running the 4 arithmetic adapter synthesis tasks that timed out after 24 hours with the hand-modified non-branching `median` function

Table 3.2 clearly shows that the non-branching version of `median` is much faster at finding the "insubstitutable" conclusion. While a significant portion of the total running time is still used by the last adapter search step, the non-branching version of `median` significantly accelerates the FuzzBALL-based counterexample search too. These results present the need for integrating path-merging as a feature in a symbolic executor since it not only alleviates path explosion but also makes adapter synthesis usable.

3.3 Conclusion

Path-merging provides users of adapter synthesis tools the flexibility to specify adaptation operators at the source-level. It also allows the possibility of even more powerful adaptation operators being used than the ones presented in chapter 2. Path-merging provides further performance benefits to symbolic execution as shown by the results in the previous section. In the following chapter, we present the reinterpretation of path-merging ideas for a Java symbolic executor and demonstrate its benefits.

Chapter 4

Java Ranger: Static Region Summaries For Efficient Symbolic Execution Of Java

4.1 Introduction

Symbolic execution is a popular analysis technique that performs non-standard execution of a program: data operations generate formulas over inputs, and branch constraints along an execution path are combined into a predicate. Originally developed in the 1970s [13, 14], symbolic execution is a convenient building block for program analysis, since arbitrary query predicates can be combined with the logical program representation, and solutions to these constraints are program inputs illustrating the queried behavior. Some of the applications of symbolic execution include test generation [90, 91], equivalence checking [15, 92], vulnerability finding [93, 94], and protocol correctness checking [95]. Symbolic execution tools are available for many languages, including CREST [96] for C source code, KLEE [97] for C/C++ via LLVM, JDart [98] and Symbolic PathFinder (SPF) [99] for Java, and S2E [100], FuzzBALL [101], and `angr` [94] for binary code. Some of these tools, such as `angr` and Mayhem [102] that operate at the binary-level, are used for finding Others, such as KLEE, are used for exploring system-level programs for software engineering purposes. Other tools such as

Symoblic Pathfinder are used for detecting errors in complex flight control software [103].

Although symbolic analysis is a popular technique, scalability is a substantial challenge for many applications. In particular, symbolic execution can suffer from a *path explosion*: complex software has exponentially many execution paths, and baseline techniques that explore one path at a time are unable to cover all paths. Dynamic state merging [104, 105] provides one way to alleviate scalability challenges by opportunistically merging dynamic symbolic executors, effectively merging the paths they represent. Avoiding even a single branch point can provide a multiplicative savings in the number of execution paths, though at the potential cost of making symbolic state representations more complex.

Veritesting [106] is another technique that can dramatically improve the performance of symbolic execution by effectively merging paths. Rather than explicitly merging state representations, veritesting encodes a local region of a program containing branches as a disjunctive region for symbolic analysis. This often allows many paths to be collapsed into a single path involving the region. In previous work [106], constructing bounded static code regions was shown to allow symbolic execution to find more bugs, and achieve more node and path coverage, when implemented at the X86 binary level for compiled C programs. This motivates us to investigate using static regions for symbolic execution of Java software (at the Java bytecode level). We define a multi-path region as a part of the code that begins at a conditional branch instruction and ends at the program location where all paths through it are merged. Figure 4.1 shows two examples of regions of code highlighted in green. The first region on line 7 of Figure 4.1 shows a region with up to two paths through it modifying a local variable named `inWord`. The second region spans lines 9–16 and it adds `1` to a `wordCount` variable if a delimiter value of `0` is found in `list`.

There are substantial differences between compiled Java programs and C programs. In C programs, most functions are statically dispatched and exceptions do not occur, allowing C compilers to inline code and create relatively large code regions without non-local jumps. In Java programs, most functions are dynamically dispatched, methods tend to be small, and the compiler assumes an "open world" so most functions are not inlined. In addition, many, if not most, Java bytecodes can throw exceptions, leading to many small, dynamically dispatched regions with many non-local control jumps. In a

naive implementation, these non-local jumps reduce the size of summaries that can be created and increase the branching factor for exploration, leading to poor performance. This makes Java more challenging for creating static summaries.

To yield good performance, we must resolve many of the potential non-local jumps at region instantiation time: determining whether exceptions are possible, collapsing `return` instructions, and inlining dynamically dispatched method summaries [107].

A common feature of Java code at the boundary of path merging is *exceptions*. If an exception can potentially be raised in a region, the symbolic executor needs to explore that exceptional behavior. But, it is possible for other unexceptional behavior to also exist in the same region. For example, it can be in the form of a branch nested inside another branch that raises an exception on the other side. Summarizing such unexceptional behavior while simultaneously guiding the symbolic executor towards potential exceptional behavior reduces the branching factor of the region.

A third common feature of Java at the frontier of path merging is the *return* instruction. For a region summary, a return instruction represents an *exit point* of the region. An exit point is a program location in the region at which paths in the region have been merged into a single path. If region has multiple exit points in the form of multiple return instructions, each predicated on a condition, the symbolic executor can construct a formula that represents all return values predicated on their corresponding conditions. Summarizing such multiple control-flow returning exit points of a region into a single exit point further reduces the branching factor of the region.

While summarizing higher-order regions, finding single-path cases, and converting multiple returning exit points into a single returning exit point is useful to improve scalability, representing such summaries in an intermediate representation (IR) that uses static single-assignment (SSA) form provides a few key advantages. (1) It allows region summaries to be constructed by using a sequence of transformations, with each transformation extending to add support for new features such as heap accesses, higher-order regions, and single-path cases. (2) It allows for simplifications such as constant propagation, copy propagation, constant folding to be performed on region summaries. (3) It makes the construction of region summaries more accessible to users of the symbolic execution tool, thereby making path merging more useful.

In summary, we make the following contributions in this paper:

(1) To improve performance, we create *higher-order* regions that take other regions as arguments: this allows us to "just in time" instantiate regions containing dynamic dispatch calls, once types are known.

(2) Next we must efficiently handle non-local jumps introduced by exceptions and returns. We propose a technique named Single-Path Cases for splitting regions into exceptional and non-exceptional outcomes, and use Standard Symbolic Execution (SSE) for the exceptional cases. To make this efficient, at instantiation, we use syntactic techniques to remove most of the possible exceptions.

(3)We collapse multiple returns into one and use standard symbolic execution to execute the `return` instruction returning the summarized return value.

In our experiments, we demonstrate exponential speedups on benchmarks (in general, the more paths contained within a program, the larger the speedup) over the unmodified Java SPF tool using this approach.

This chapter extends our initial investigations of Java veritesting reported in a paper at the 2017 JPF workshop [108]. Our workshop paper motivated some of the ways in which Java veritesting is different from veritesting for binary code and the value that veritesting could provide, but it did not describe an end-to-end automated implementation. The present paper describes a number of conceptual improvements that are new since the workshop paper, including the Ranger IR, single-path cases, and the details of our higher-order region approach. We have also made a number of architectural changes, such as switching from Soot to Wala for static analysis and using Green for formula representation, and we have integrated Java Ranger as an extension to SPF and evaluated it in several case studies.

4.1.1 Motivating Example

```
1  List<Integer> list = new ArrayList<>(200);
2  // put 200 symbolic int into list
3  int wordCount = 0;
4  boolean inWord;
5  if (list.size() > 0) {
6    int firstElement = list.get(0);
7    if (firstElement == 0) inWord = false; else inWord = true;
8    for (int i = 0; i < list.size(); i++) {
9      if (inWord) {
10     // list.get(i) returns sym. int
11       if (list.get(i) == 0) {
12         ++wordCount; inWord = false;
13       }
14     } else {
15       if (list.get(i) != 0) inWord = true;
16     } } }
17 return wordCount;
```

region 1
region 2

Figure 4.1: An example where Java Ranger summarizes two multi-path regions

Consider the example shown in Figure 4.1. The code counts the number of words where the concrete value `0` acts as a delimiter for words. `list` refers to an `ArrayList` of 200 `Integer` objects which have an unconstrained symbolic integer as a field. Counting the number of words in this example requires 2^{200} execution paths using standard symbolic execution with one branch introduced per loop iteration. However, we can avoid this path explosion by merging the two paths arising out of the `list.get(i) == 0` branch. Such path merging requires us to compute a summary of all behaviors arising on both sides of the branch of the if-statement at line 11 in Figure 4.1.

Constructing such a summary for this simple region is not straightforward. The call to `list.get(int)` is actually a call to `ArrayList<Integer>.get(int)` which internally does the following: (1) It checks if the index argument accesses a value within bounds of the `ArrayList` by calling `ArrayList<E>.rangeCheck(int)`. If this access is not within bounds, it throws an exception. (2) It calls `ArrayList<E>.elementData(int)` to access an internal array named `elementData` and get the entry at position `i`. This call results in an object of class `Integer` being returned. (3) It calls `Integer.intValue()` on the

object returned by the previous step. This call internally accesses the `value` field of `Integer` to return the integer value of this object.

The static summary of `ArrayList<Integer>.get(int)` needs not only to include summaries of these three methods but also to include the possibility of an exception being raised by the included summary of `ArrayList<E>.rangeCheck(i)`. The method whose summary is to be included depends on the dynamic type of the object reference on which the method is being invoked. In this example, the dynamic type of `list` is `ArrayList`, whereas it is declared statically as having the type `List`. Our contributions to path-merging uses the runtime type of `list` to inline such method summaries to merge path all 2^{200} paths into a single execution path in this example. We walk through the transformations that allow such path merging on this example in Section 4.3.

4.2 Related Work

Path explosion hinders scalable use of symbolic execution, so an appealing direction for optimization is to combine the representations of similar execution paths, which we refer to generically as *path merging.* If a symbolic execution tool maintains objects representing multiple execution states, a natural approach is to merge these states, especially ones with the same control-flow location. Hansen et al. [104] and Kuznetsov et al. [105] are representative examples of this approach. Kuznetsov et al. take the approach even further by not just merging objects at the same control-flow location, but also checking if the merge is beneficial for further symbolic execution of the program. A larger departure in the architecture of symbolic execution systems is the MultiSE approach proposed by Sen et al. [109], which represents values including the program counter with a two-level guarded structure, in which the guard expressions are optimized with BDDs. Another approach to achieve path merging is to statically summarize regions that contain branching control flow. This approach was proposed by Avgerinos et al. [106] and dubbed "veritesting." A veritesting-style technique is a convenient way to add path merging to a symbolic execution system that maintains only one execution state, like SPF. Avgerinos et al. implemented their veritesting system MergePoint to apply binary-level symbolic execution for bug finding. They found that veritesting provided a dramatic performance improvement, allowing their system to find more bugs

and have better coverage. The static regions used by MergePoint are intra-procedural, but they can have any number of "transition points" at which control can be returned to regular symbolic execution. Avgerinos et al. do not provide details about how MergePoint represents memory accesses or integrates them with veritesting, though since MergePoint was built as an extension of the same authors' Mayhem system, it may reuse techniques such as symbolic representation of loads from bounded memory regions [102].

The veritesting approach has been integrated with another binary level symbolic execution engine named `angr` [94]. However angr's authors found that their veritesting implementation did not provide an overall improvement over their dynamic symbolic found, they observed that giving more complex symbolic expressions slowed down the SMT solver enough that total performance was degraded. We have also observed complex expressions to be a potential cost of veritesting, but we believe that optimizations of the SMT solver interface and potentially heuristics to choose when to use static regions can allow them to be a net asset.

The way that Java Ranger and similar tools statically convert code regions into formulas is similar to techniques used in verification. In the limit where all relevant code in a program can be summarized, such as with WBS and TCAS in Section 4.4, Java Ranger performs similarly to a bounded symbolic model checker for Java. SPF and Java Ranger build on Java Pathfinder (JPF) [110], which is widely used for explicit-state model checking of Java. The most closely related Java model checking tool is JBMC [111], which shares infrastructure with the C tool CBMC. JBMC performs symbolic bounded model checking of Java code, transforming code and a supported subset of the standard library into SMT or SAT formulas that represent all possible execution paths. The process by which JBMC transforms its internal code representation into SMT formulas is similar to how Java Ranger constructs static regions. However, the dynamic dispatch aspects of Java make creating entirely static representations expensive. We believe that our approach can yield simpler SMT formulas in many cases where it is difficult to completely statically summarize program behavior, and can be used in cases when software is too large and/or complex to be explored completely. We are interested in performing an experimental comparison between Java Ranger and JBMC, but our current benchmarks may not be suitable because they do not include

properties to check. We instead compare Java Ranger with JBMC using the latest set of benchmarks used in the Java category of the Software Verification Competition 2020.

A wide variety of other enhancements to symbolic execution have been proposed to improve its performance, including caching and simplifying constraints, summarizing repetitive behavior in loops, heuristic guidance towards interesting code, pruning paths that are repetitive or unproductive, and many domain-specific ideas. A recent survey by Baldoni et. al. [112] provides pointers into the large literature. One approach that is most related to our higher-order static regions is the function-level compositional approach called SMART proposed by Godefroid [113]. SMART differs in being based on single-path symbolic execution instead of static analysis, and targeting C it does not address dynamic dispatch.

Like Java Ranger's function summaries, SMART summarizes the behavior of a function in isolation from its calling context so that the summary can be reused at points where the function is used. But SMART uses single-path symbolic execution to compute its summaries, whereas Java Ranger uses static analysis: this makes Java Ranger's summary more compact at the expense of requiring more reasoning by the SMT solver. Because SMART was implemented for C, it does not address dynamic dispatch between multiple call targets.

A few related techniques that can accelerate symbolic execution of code are loop summarization and value set analysis. Loop-extended symbolic execution introduced partial loop summarization by having symbolic variables that represent the number of times each loop executes. This technique allowed symbolic variables to incorporate loop dependent effects along with data dependencies from program inputs.

Since path-merging makes use of static analysis to automatically summarize code, it is also susceptible to the limitations faced by Java static analysis. Two key limitations are finding which reflective method call is being used and dynamic class loading. TamiFlex [114] is one static analysis tool that can provide an answer to both limitations. It is sound with respect to a set of previously seen program runs. Integrating veritesting runs into similar problems, and using techniques from TamiFlex would allow static predicate construction beyond exit points caused by reflection or dynamic class loading.

4.3 Technique

Java Ranger operates over regions of Java bytecode for which there is an opportunity for path merging. In particular Java Ranger distinguish three types of regions. (1)*Multi-Path Region*: this corresponds to the Java bytecode of an acyclic subgraph of the control-flow graph (CFG). It begins from the basic block containing a conditional branch and ends at this basic block's immediate post-dominator. Since Java allows `if-then-else` statements to be nested, a multi-path region may also contain other multi-path regions, (2)*Method Region*: this corresponds to the Java bytecode of a method, and (3) *Higher-order region*: this corresponds to a multi-path region that contains at least one method invocation.

In this section we start by introducing JR's grammar, then discuss how JR recovers statements from a CFG, next we introduce JR's static-summary instantiation algorithm, finally we discuss correctness of Java Ranger.

4.3.1 Java Ranger Grammar

$\langle val \rangle$ ::= C | Z | B $\langle var \rangle$::= .. | Id_s | Id_{sr} $\langle ref \rangle$::= Id $\langle field \rangle$::=Id $\langle class \rangle$::= Id

$\langle sig \rangle$ ::= method-signature $\langle summary \rangle$::=($\overrightarrow{x}$.s) $\langle exp \rangle$::= ... | $\gamma(\langle exp \rangle, \langle exp \rangle, \langle exp \rangle)$

$\langle stmt \rangle$ ::= ... | skip | if $\langle$exp$\rangle$ then $\langle$stmt$\rangle$ else $\langle$stmt$\rangle$ | $\phi(\langle \text{var} \rangle, \langle \text{var} \rangle, \langle \text{var} \rangle)$
| invoke($\langle var \rangle, \langle ref \rangle, \langle sig \rangle, \langle \overrightarrow{exp} \rangle$) | return $\langle exp \rangle$ | putfield($\langle ref \rangle, \langle field \rangle, \langle exp \rangle$)
| getfield($\langle var \rangle, \langle ref \rangle, \langle field \rangle$) | aload($\langle var \rangle, \langle ref \rangle, \langle exp \rangle$) | astore($\langle ref \rangle, \langle exp \rangle, \langle exp \rangle$)
| new($\langle$class$\rangle, \langle \overrightarrow{e} \rangle$) | throw $\langle$e$\rangle$ | single_exit| return_exit

Figure 4.2: Main Constructs in Ranger IR

Figure 4.2 shows grammar of JR IR. Java Ranger supports variables of types, `char`, `int`, and `boolean` as used in Java. Java Ranger IR supports symbolic variables represented by Id_s used to capture symbolic expressions. It uses a special return-symbolic variable Id_{sr} to capture the summarization of return expressions. Java Ranger IR identifies object references by ref, field accesses by $field$, classes by $class$, and method signatures by sig.

JR has a special construct to identify a *static summary* ($\overrightarrow{x}$.s), which describes

the statically recovered statement s from the CFG of a multi-path or a method region along with its input $\overrightarrow{x}$. Later, JR may *instantiate* these static summaries. The goal of the instantiation process, described in Algorithm 1, is to convert a static summary to its *fully-linearized* form, which is a form of the summary that can be written back to the dynamic context of the region as a constraint on the path condition. This process substitutes inputs from the Standard Symbolic Execution (SSE) context at a given point and rewrites summaries through transformations.

Java Ranger extends expressions generated from standard symbolic execution with a γ-expression: $\gamma(e_1, e_2, e_2)$, where e_1 is the condition and e_2 and e_3 are the returned expression if the condition is true or not respectively. Note that a recovered JR IR statement, might contain γ-expression for merging writes to local variables. A γ-expression implements Gated Single Assignment (GSA) [115] to a variable.

However for field and array references changes, JR needs to create the right γ-expression for their updates. This happens during field and array transformation.

Statements include assignment, composition, **skip**, **if**-statements, ϕ-statement: $\phi(x_3, x_1, x_2)$ to capture joins of local variable changes of x_1 and x_2 into x_3, method invocation: **invoke**(x,r,g,$\overrightarrow{e}$) where x is the output of the invocation for a method g of reference r and actual parameters $\overrightarrow{e}$, return statement return e, putfield: **putfield**(r, f, e) to put the value of expression e in the field f of reference r, **getfield**(x, r, f) where the output value is stored in x. Similar statements are for supporting loads from and stores to an array: **aload**(x, r, I), **astore**(r, I, e) where x contains the output of the load, I is the index expr and e is the expression to be stored, object creation: **new**(e,$\overrightarrow{e}$), throw exception **throw**(e) and finally **single_exit** and **return_exit** to mark *exit-points*.

An exit-point is program location at which JR transfers execution back to SSE. There are three kinds of exit points in Java: a program location that corresponds to the conditional branch's immediate post-dominator, a program location that performs a non-local jump in the form of a `return` instruction, and a set of program locations that throw exceptions. We refer to these three exit points as a non-exceptional and non-returning exit point (single_exit), a returning exit point (return_exit), and a exceptional exit point (throw(e)) respectively in the rest of this work.

4.3.2 IR Statement Recovery

While Java bytecode uses registers (stack slots for local variables) and the stack for instruction operands, the recovered Ranger IR captures the multi-path or method region in Gated Single Assignment (SSA) form. Since the analysis used in this step happens just-in-time, we refer to it as *jit analysis*. The algorithms of jit-analysis are similar to those used for decompilation [116]. Starting from an initial basic block in a control-flow graph recovered by Wala [117], the algorithm first finds the immediate post-dominator of all *normal* control paths, that is, paths that do not end in an exception or return instruction. It then looks for nested self-contained subgraphs.

The algorithm systematically attempts to build static summary for every branch instruction, even if the branch is already contained within another multi-path region. The reason is that, it may not be possible to instantiate the larger multi-path region depending on whether summaries can be found for the *dynamically-dispatched* functions, and whether references are *uniquely determinable* for region outputs.

Local inputs and outputs are also determined during this process. In particular, given a Ranger IR statement, the first *use* of a stack slot is deemed a local input and the last *def* of a stack slot is deemed a local output.

For region 2 shown in the motivating example in Figure 4.1, the recovered statement is shown below where $x58$ corresponds to input from $inWord$, and $x54$, $x55$ correspond to outputs to $wordCount$, $inWord$ respectively.

```
if (!(x58 == 0 )) then {
  x44:=invoke(x9,get(I)Ljava/lang/Object,x59);
  x47:=invoke(x44,intValue()I,_);
  if (!(x47 != 0))   x48 := (x57 + 1) } else { ... }
x54 := (γ !(x58==0) (γ !(x47!=0) x48 x57) (γ !(x53==0) x57 x57));
x55 := (γ !(x58==0) (γ !(x47!=0) 0 x58) (γ !(x53==0) 1 x58));
```

4.3.3 Static-Summary-Instantiation

Algorithm 1 describes how JR instantiates a summary. It starts when SSE is about to execute a conditional branch instruction with symbolic operands for which JR has a

static summary s of the form ($\overrightarrow{x}$.y). JR creates an initial ranger environment ω_o from the current SSE state δ_{sym} by populating its inputs.

Algorithm 1: JR Static-Summary-Instantiation Algorithm

```
1  Input: (Ranger IR Statement s, SSE δ_sym)
2  ω_o = construct-initial-state(s, δ_sym); (ω_er, s_er) = eliminate early-return(ω_o, s)
3  (ω_α, s_α) = α-renaming(ω_α, s_α);   (ω_bf, s_bf) = (ω_α, s_α);   (ω_af, s_af) = null
4  repeat
5      repeat
6          if (ω_af, s_af) ≠ null then ω_bf, s_bf = (ω_af, s_af);
7          (ω_bf, s_sub) = substitute local inputs(ω_bf, s_bf);
8          (ω_hg, s_hg) = inline higher-order(ω_bf, s_sub);
9          (ω_f, s_f) = create ref. ssa(ω_hg, s_hg); (ω_ar, s_ar) = create arr. ssa(ω_f, s_f);
10         (ω_smpl, s_smpl) = simplify (ω_ar, s_ar);   (ω_af, s_af) = (ω_smpl, s_smpl);
11     until (ω_bf, s_bf) = (ω_af, s_af)
12     (ω_af, s_af) = inline higher-order(ω_af, s_af);
13 until ((ω_bf, s_bf) = (ω_af, s_af))
14 (ω_sp, s_sp) = collect single-path cases(ω_af, s_af); (ω_sp, s_ln) = linearize(ω_sp, s_sp);
15 if is-fully-linearized(ω_sp, s_ln) then
16     e = generate const.(ω_sp, s_ln); populate outputs (ω_sp, e); skip region exec.;
17 else abort;                                   /* resume SSE from cond. branch */
```

Next, Java Ranger runs different transformations on the multi-path summary (lines 4-14) with the goal of reducing the multi-path's IR statement into a fully-linearized form. These transformations perform different operations on the multi-path summary, such as inlining method summaries. Because of the interaction between dynamic dispatch and simplification, it is necessary to iteratively transform the static summary until a fixed-point is reached. For example, the type of an object may not be known until we summarize field accesses, which we can then inline, which may lead to additional field accesses that must be summarized. JR uses a nested fixed-point computation to ensure that a maximal number of regions can be linearized. If the transformations produce a fully-linearized form of the multi-path summary, then Java Ranger adds the constraint of the multi-path summary to the path condition, populates the instantiated summary's outputs, and sets the program counter (the next instruction to be executed) to the

address of the instruction that is at the beginning of the immediate post-dominator. Otherwise, JR aborts and allows Standard Symbolic Execution (SSE) to resume execution (lines 15-17).

In the following section we describe each transformation in more detail. Readers interested in the semantics of some of JR's transformations are referred to [118].

Eliminate Early-Returns: In this transformation, JR identifies return-instructions inside conditional branches, we call those return instructions *early-returns*. JR converts static summaries with early-return into a summary that assigns a symbolic return variable. This return-variable is assigned to a γ-expression that evaluates to different return values, each predicated on the condition that would cause that value to be returned.

Alpha-renaming: This transformation helps distinguish Ranger IR variables across multiple instantiations of the same static summary by adding a subscript unique to each instantiation of a static summary.

Substitute Local Variable Inputs: During this transformation, local variable inputs in the static summary are substituted with values obtained from SSE context. Running this transformation on the previous Ranger IR statement causes $x9$ to be substituted by 375 where *$2* is added by α-renaming.

```
x44$2 := invoke(375, get(I)Ljava/lang/Object,x59$2, virtual);
```

Inline Higher-order Regions: This transformation instantiates the summary for a higher-order region in four steps. (1) The called method's summary is retrieved, and alpha-renaming is run on it. (2) Parameters passed to the method are substituted in the method summary by applying local variable substitution. (3) The previous two steps are run again if other method invocations are present in the method summary. (4) The resulting method summary is used to replace the method call in the summary of the higher-order region.

As an example of the effects of this transformation, the summary for $ArrayList.get(I)java.lang.Object$ is inlined in the multi-path region's summary to get the following Ranger IR statement. $x58$ in recovered static summary has been substituted by $x40\$1$ because it was read as a local variable input ($inWord$) that had the

symbolic value $x40\$1$ (a consequence of summarizing region 1 in Figure 4.1).

```
if (!(x40$1==0)) then {
  x4$4 := get_field(375,size);
  if (!(0 < x4$4 )) Throw Instruction;
  x4$5 := get_field(375, elementData);
  x5$5 := array_load(x4$5,0);
  x6$3 := x5$5;
  x44$2 := x6$3;
  x47$2 := invoke(x44$2, intValue()I,_);
  if ((x47$2 != 0)) x48$2 = (0 + 1);
} else { ... }
x54$2 := (γ !(x40$1==0) (γ !(x47$2!=0) x48$2 0) (γ !(x53$2==0) 0 0));
x55$2 := (γ !(x40$1==0) (γ !(x47$2!=0) 0 x40$1) (γ !(x53$2==0) 1 x40$1));
```

Create Field References SSA: This transformation summarizes field accesses in the multi-path region. It constructs a new field access variable for every assignment to a field using a concrete object reference in the instantiated summary. This new field access variable uses two monotonically increasing subscripts: (1) a path subscript p to distinguish between assignments to the same field on the same execution path, (2) a global subscript G to distinguish between assignments to the same field across execution paths. At the non-exceptional and non-returning exit point of the instantiated summary, field assignments to the same field are merged using Gated Single Assignment (GSA) [115]. Rule $\texttt{put-field}_\texttt{r}$ describes creation of fresh symbolic variable rf_{n+1}, with an incremented subscript, and adding the new variable to both the global and the path subscript maps. A put-field is then rewritten to an assignment statement in the updated environment ω'. To continue expanding the motivating example, the field references transformation on the above Ranger IR statement causes the values 200 and 397 to be assigned to $x44 and x45$ respectively.

Simplify: This transformation runs constant propagation, copy propagation, and constant folding [119]. The simplification transformation maintains variable-to-constant mapping in the instantiated summary's environment (not shown). This causes the assignment to $x48\$2$ to be removed from the below Ranger IR statement and After running field transformation and simplification, $x48\$2$ is substituted by the constant 1 as shown

below.

```
if (x40$1 != 0) then {
  x5$5 := array_load(397, 0);  x45$2 := x5$5;
  x47$2 := invoke(x5$55, intValue()I, _)} else { ... }
x54$2 := (γ x40$1!=0 (γ x47$2==0 1 0) 0);
x55$2 := (γ x40$1!=0 (γ x47$2==0 0 x40$1) (γ x53$2!=0 1 x40$1));
```

Create Array References SSA: This transformation translates array accesses to Ranger IR. It maintains a path-specific copy of every array when it is first accessed using a concrete array reference within a instantiated summary. Reads and writes of arrays are then done on a path-specific copy of the array. All array copies are merged at the non-exceptional and non-returning exit point of the instantiated summary. The merged array copy represents array outputs of the instantiated summary. Out-of-bounds array accesses are explored as an exceptional exit point. The effect of this transformation on the last shown Ranger IR statement produces the following where the value at index 0 in array reference 397 was 380 and the length of array at reference 397 was 200.

```
if (x40$1 != 0) then {
  if ((0 < 200) && (0 >= 0)) x5$5 := 380;  else Throw Instruction
  x45$2 := x5$5; x47$2 := invoke(x5$5, intValue()I, _)} else { ... }
x54$2 := (γ x40$1!=0 (γ x47$2==0 1 0) 0);
x55$2 := (γ x40$1!=0 (γ x47$2==0 0 x40$1) (γ x53$2!=0 1 x40$1));
```

Collect Single-Path Cases: In this transformation, Java Ranger identifies exit statements in the instantiated summary's Ranger IR statement, collects their path predicates, and prunes them away from the Ranger IR statement. The outcome of this process is: (a) Ranger IR statement that captures non-exceptional behavior in the instantiated summary and (b) a predicate is used to explore the summary's exceptional exit point.
Linearize: JR removes if-then-else statements since their condition is already inside the γ- expression. A fully-linearized statement in Ranger IR only includes composition and assignment over fully-linearized expressions. Rule `if-linear` shows rewriting of if-statements into composition of statements.
Running this transformation (after another simplification and inlining of the summary

of $Integer.intValue()$) produces:

```
x54$2 := (γ x40$1!=0 (γ x1==0 1 0) 0);
x55$2 := (γ x40$1!=0 (γ x1==0 0 x40$1) (γ x1!=0 1 x40$1));
```

Note that the *value* field accessed by $Integer.intValue()$ in the object referenced by the value 380 was set to the symbolic integer $x1$. The variables $x542 and x552$ correspond to outputs to $wordCount$, $inWord$ in Figure 4.1 respectively.

Generate Constraint: This transformation translates the fully linearized Ranger IR statement to a constraint in Green [120]. This is done by translating statement composition into conjunction, assignments as equality constraints with assignments of γ-expression being translated as a disjunctive equalities.

4.3.4 Empirical Verification of Java Ranger summaries

```
1  public class EqChkHarness {
2    void testHarness(TestRegionBaseClass v,
3                     int int_inputs[], boolean bool_inputs[], char char_inputs[]) {
4      Outputs outSPF =
5        v.testFunctionWOPM(int_inputs, bool_inputs, char_inputs); // path-merging
       turned off
6      Outputs outJR =
7        v.testFunction(int_inputs, bool_inputs, char_inputs); // path-merging turned
       on
8      if (outSPF.equals(outJR))
9        System.out.println("Match");
10     else {
11         // executed only when the path-merging output does not match
12         // output generated by the standard symbolic executor
13         System.out.println("Mismatch");
14         assert(false);
15   }
16 }
```

Listing 4.1: Test harness to compare the side-effects of instantiated summaries used by Java Ranger with side-effects generated by executing the same multi-path region with standard symbolic execution

Java Ranger produces static summaries for multi-path regions of code and instantiates them when symbolic execution reaches the multi-path region of code. Instantiating the static summary allows Java Ranger to resume standard symbolic execution directly from the exit points of the multi-path region. While this provides Java Ranger a significant performance benefit, it is crucial to preserve the semantics of the multi-path region in the instantiated summary. Excluding any side-effects of the multi-path region from the instantiated summary breaks the soundness of the underlying static symbolic executor. Therefore, we empirically validated the correctness of Java Ranger's instantiated summaries by using equivalence-checking.

```
1  public abstract class TestRegionBaseClass {
2    // Classes that wish to test a path-merging in a multi-path region
3    // for correctness should inherit from TestRegionBaseClass and
4    // put the multi-path region inside their overridden testFunction
5    abstract Outputs testFunction(int int_inputs[],
6                                  boolean bool_inputs[],
7                                  char char_inputs[]);
8
9    // Trapdoor method to make Java Ranger turn path-merging off for callees
10   Outputs testFunctionWOPM(int int_inputs[],
11                                 boolean bool_inputs[],
12                                 char char_inputs[]) {
13     return testFunction(int_inputs, bool_inputs, char_inputs);
14   }
15 }
```

Listing 4.2: Design of the base class used together with the test harness presented in Listing 4.1

Equivalence-checking is a technique that uses symbolic execution to compare two fragments of code for semantic equivalence. It runs the two fragments of code in the same execution context with the same inputs and compares their side-effects for equality. To check Java Ranger's instantiated region summaries for equivalence with the semantics of the multi-path region, we employed the test harness presented in Listing 4.1. The companion class, `TestRegionBaseClass`, is provided in Listing 4.2. To set up the comparison of the outputs of an instantiated summary of a multi-path region with the same outputs

generated by a standard symbolic executor, the code for the multi-path region has to first be added to the `testFunction` method of a subclass of `TestRegionBaseClass`. This subclass should then call `testHarness` method of `EqChkHarness` which will first call the subclass'overridden `testFunction` method via `testFunctionWOPM`. The `testFunctionWOPM` method is a special method that makes Java Ranger turn off path-merging for it and its callees. Next, the test harness will call the subclass'overriden `testFunction` directly. Finally, the testharness will compare both the outputs for equality as shown on line 8 of Listing 4.1. We constructed 29 different test cases of multi-path regions that consisted of path-merging across writes to local variables, static and non-static fields, arrays, higher-order regions with depth ranging from 1 to 5, recursive higher-order regions, multi-path regions with exceptional exit points, and higher-order regions with multiple return instructions. For each of these 29 test cases, the instantiated summary used by Java Ranger produced outputs equivalent to the outputs generated by the standard symbolic executor.

4.4 Evaluation

4.4.1 Implementation

We implemented Java Ranger as an extension of Symbolic PathFinder [99] tool. We used the existing *listener* framework that will invoke a callback function for each bytecode instruction executed by SPF. JR adds a listener to SPF that, on every symbolic branch, attempts path merging as described in Algorithm 1. We used the control-flow graph recovered by Wala [117] to bootstrap our static statement recovery process. While we found our static statement recovery was capable of summarizing thousands of regions in Java library code, many of these summaries could not be instantiated due to JPF's use of native peers [121]. To avoid these unnecessary instantiation failures and target our static statement recovery towards the benchmark code, we turned off statement recovery across a few Java library packages in Wala on all benchmarks. JR uses the incremental solving mode of Z3 [122] with the bitvector theory. The incremental solving mode provides only the last constructed constraint to the solver instead of passing the entire path condition every time a query is to be solved. The incremental solving mode significantly reduces the number of times a constraint has to be passed to the solver.

Finally, JR uses a heuristic to estimate the number of paths through a fully linearized summary. The fully linearized summary is used only if the estimated number of paths in the multi-path region is greater than the number of exit points in it.

4.4.2 Experimental Setup

We sought answers to the following research questions.

RQ1: *Does Java Ranger reduce the number of paths?*

RQ2: *Does Java Ranger reduce the time required for symbolic execution?*

RQ3: *How much does each Java Ranger feature contribute to performance?*

We evaluated the performance of Java Ranger using the nine benchmarking programs presented in Table 4.1. The first eight were provided by Wang et al. [123] and the last one (MerArbiter) by Yang et al. [124]. We used SPF as the Standard Symbolic Executor and compared the performance of JR with SPF when exploring all feasible paths through a benchmark. Path merging is useful in symbolic execution when it is used for checking a property on all feasible behaviors or finding all bugs in a program. Since exploration of all feasible behaviors through a program is a lower bound on running time for such applications of symbolic execution, we compared JR with SPF when exploring all feasible paths through each benchmark. None of these benchmarks performed multi-threaded execution.

Bench mark name	Description	SLOC	# classes	# methods
WBS	synchronous reactive component developed to make aircraft brake safely when taxing, landing, and during a rejected take-off	265	1	3
TCAS	Siemens suite program to maintain altitude separation between aircraft	300	1	12
replace	Siemens suite program to search for a pattern in input and replace it with another string	795	1	19
Nano XML	XML Parser	4610	17	129
Siena	Internet-scale event notification middleware for distributed event-based applications	1256	10	94
Schedule	priority scheduler	306	4	27
Print Tokens2	lexical analyzer	570	4	30
ApacheCLI	command-line parser	3612	18	183
Mer Arbiter	flight software component of NASA JPL Mars Exploration Rovers	4697	268	553

Table 4.1: Benchmark programs used to evaluate Java Ranger

RQ1, RQ2: We ran every benchmark with SPF and JR with the most number of symbolic inputs that would finish in a 12 hour time budget. We ran all of our experiments on a machine running Ubuntu 16.04.6 LTS with Intel(R) Xeon(R) CPU E5-2623 v3 processor and 192 GB RAM. We report our results in Table 4.2. JR achieves an average 23% and 66% reduction in total running time and number of execution paths respectively across all benchmarks. It also achieves an average of 63% reduction in the number of solver queries.

Bench mark name	#sym input	tool	total time (sec)	% red. in time	static analysis time (sec)	# exec. paths	%red. in # exec. paths	% red. in # que- ries	# summ. used
WBS	15	SPF	4427.7	99.9	0.0	7.96E+06	100	100.00	-
	30	JR	4.2		2.3	1.00E+00			140
TCAS	24	SPF	353.1	99.1	0.0	3.92E+04	100	100.00	-
	120	JR	3.1		1.7	1.00E+00			40
replace	11	SPF	1145.3	-187.0	0.0	7.57E+05	88.1	67.30	-
	11	JR	3287.6		5.9	9.04E+04			6502
Nano XML	7	SPF	5741.4	46.2	0.0	3.61E+06	84.6	81.00	-
	7	JR	3087.1		3.1	5.54E+05			147185
Siena	6	SPF	5571.9	-2.6	0.0	2.99E+06	0	0.00	-
	6	JR	5715.6		7.3	2.99E+06			0
Schedule	3	SPF	1.5	-70.3	0.0	3.43E+02	0	0.00	-
	3	JR	2.5		3.4	3.43E+02			0
Print Tokens2	5	SPF	17045.8	21.3	0.0	3.06E+06	40.4	38.70	-
	5	JR	13421.3		25.1	1.82E+06			1981982
Apache CLI	5+1	SPF	4121.4	45.7	0.0	2.48E+05	92.9	99.10	-
	5+1	JR	2238.1		5.3	1.76E+04			168907
Mer Arbiter	24	SPF	9494.0	80.3	0.0	2.53E+05	83.9	81.50	-
	24	JR	1873.4		3.6	4.08E+04			59845

Table 4.2: Comparing execution time and path count between JR and SPF

JR achieves a significant speed-up over SPF with 5 (WBS, TCAS, NanoXML, ApacheCLI, MerArbiter) of the 9 benchmarks in running time and number of execution

paths. JR achieves a modest 21% and 40% reduction in running time and number of execution paths respectively with the PrintTokens2 benchmark.

JR statically summarizes the entire step functions of WBS and TCAS. This step functions of WBS and TCAS take 3 and 12 symbolic inputs respectively. In WBS, JR summarizes multi-path regions with deeply nested `if` bytecode instructions, in one case summarizing a multi-path region 9 branches deep. JR inlines 28 method summaries in each step of TCAS, many of which summarize multiple return values into a single formula that represents all the return values of the method. While SPF does not finish more than 5 steps of WBS and 2 steps of TCAS within 24 hours, JR finishes 10 steps of both benchmarks within 2.81 seconds and 1.41 seconds respectively.

Return value summarization proves to be a crucial feature for JR on the PrintTokens2 and NanoXML benchmarks. JR uses fully-linearized summaries for several multi-path regions that contain multiple `return` instructions. With PrintTokens2, JR fully linearizes static summaries for 4 multi-path regions, 2 of which involve summarizing multi-path regions with multiple `return` instructions. The NanoXML benchmark also several multi-path regions that have a `return` instruction on every path. When running NanoXML, JR is able to inline more than 8,000 method summaries containing such multi-path regions with 8 symbolic inputs and finish in about 8 hours while SPF needs about 18 hours to explore all feasible paths. Being able to summarize multiple control-flow returning exit points into a single such exit point proves to be a crucial feature in JR for this benchmark.

JR reduces the number of execution paths by about 88% in replace but incurs an increase in execution time by 187%. This increase occurs even when JR transforms 20 multi-path regions into 7334 fully-linearized summaries. On manually investigating the replace benchmark, we found that the outputs of most summaries were being branched on later causing the benefit from path-merging to be lost. We tested this hypothesis by running an automated evaluation where we restricted JR to using summaries for different fixed subsets of the full set of multi-path regions it summarized in replace. First, we determined the most number of symbolic inputs with which JR could explore all paths in replace in less than a minute using summaries for as many multi-path regions as possible. We found this number to be 6 symbolic inputs and JR instantiated summaries for 14 multi-path regions. Next, we ran JR allowing it to use summaries for

every possible subset of these 14 multi-path regions. Smaller subsets were tried first. The total number of region subsets was $2^{14} - 1$. After running this evaluation for 96 hours, we found that summaries for all possible region subsets of up to 4 regions had been run. We found that no subset containing up to 4 regions allowed JR to reduce the running time and number of execution paths in replace as compared to SPF. We hypothesize that reduction in the number of solver queries comes with an increase in solving complexity causing an increase in solving time. The 67.3% reduction in the number of solver queries causes a 240% increase in solver time spent by JR. In the future, we plan to mitigate such negative effects of path-merging by integrating JR with a query count estimation heuristic [105].

While not instantiating any summary, JR incurs a 2.6% running time overhead on Siena. This primarily results from JR's checking if a conditional branch has symbolic operands and lookup of a static summary for every symbolic branch. We restrict our results with Siena to 7 symbolic inputs because 7 is the most number of symbolic inputs for which SPF finishes exploring all feasible paths within a 24 hour time budget. Concrete branches do not cause summary instantiation. In Siena, JR begins transformation of summaries of four multi-path regions which it aborts later because it finds that the number of estimated paths through the multi-path region is equal to the number of exit points of the instantiated summaries. The total running time of Schedule is very small (2.52 seconds) compared to other benchmarks. JR's static analysis always adds about 2-3 seconds to the total time: this accounts for loading the WALA framework, constructing a class hierarchy for all classes in the classpath, and building the CFG for all methods in Wala IR. On benchmarks like Schedule with a small total running time, this overhead from static analysis is a higher percentage of the total running time. On the Schedule2 benchmark, JR fails to summarize a crucial method, `Schedule2.get_current()`, that is invoked in all summaries in Schedule2 that JR can construct. This method contains a loop whose bound is symbolic. Summarizing this method would require JR to also have the ability to summarize loops with symbolic bounds (loops that are executed for a symbolic number of iterations). Since JR currently does not summarize loops, it fails to summarize `Schedule2.get_current()` which has a cascading effect of not being able to convert summaries for these six regions into a fully linearized form.

ApacheCLI takes 9 inputs, the first 8 together form command-line options and the last input controls whether ApacheCLI should stop on encountering an invalid option. Since the 9th input is different from the first 8, we first ran ApacheCLI with the first 6 inputs and the 9th input made symbolic. We abbreviate these 6 and 1 symbolic inputs as "6+1". JR explores all feasible behaviors in ApacheCLI in about 1.5 hours whereas SPF finishes exploration in about 9 hours. SPF explores all feasible paths in ApacheCLI with "6+1" symbolic inputs in about 9 hours and does not finish with "7+1" symbolic inputs in 24 hours. But, JR can complete these explorations in about 1.5 and 15 hours respectively.

In the MerArbiter benchmark, the most significant benefit from JR comes from its ability to summarize multi-path regions with stack output. Such regions are common in Java bytecode since the JVM is both a stack machine and a register machine. All the multi-path regions that SPF needs to branch on but are summarized by JR are regions that compute a boolean value based on a symbolic branch and write it to the stack as an operand to be used by the following return instruction. In MerArbiter, most of these multi-path regions lie inside several levels of nested classes and dynamically bound field references that necessitate a fixed-point computation over the field substitution and constant propagation transformations. JR summarizes such multi-path regions with more than 282,000 instantiations needing more than 8 iterations of the fixed-point computation.

4.4.3 Comparing Java Ranger with JBMC

Since path-merging brings symbolic execution closer to symbolic bounded model-checking, we also used these benchmarks to compare JR with JBMC [125]. JBMC is a Java model checker that verifies programs by unwinding loops and looks for run-time exceptions. We ran each of our benchmarks with JBMC with the same number of non-deterministic inputs reported in the "# sym inputs" column of Table 4.2. We configured JBMC to unwind all loops in each benchmark a given number of times and also to add an assertion whose violation indicates that a loop was not unrolled sufficient times. For every benchmark, we performed binary search to find the smallest loop bound for that benchmark that would not cause a loop unwinding assertion violation with JBMC. Using this loop bound, we found JBMC was able to verify the absence of

any runtime errors in both WBS and TCAS in about a second. But, we found JBMC to be much slower with the remaining 7 benchmarks. JBMC was able to complete verification with Schedule in about 7 days. For the remaining six benchmarks, JBMC did not finish verification even though we let it run for 71 days.

4.4.4 Comparing path-merging features of Java Ranger

Benchmark name	multi-path region summ.	+higher-order regions summ.	+single-path cases	+returns summ.
WBS	0.0007	0.0007	0.0007	0.0006
TCAS	0.39	0.01	0.01	0.01
replace	1.36	2.10	2.82	2.87
NanoXML	1.31	1.28	1.54	0.54
PrintTokens2	0.88	1.02	1.02	1.02
ApacheCLI	0.17	2.54	0.50	0.54
MerArbiter	0.24	0.21	0.21	0.20

Table 4.3: Comparing the ratio of running time of Java Ranger vs SPF across the 7 benchmarks where Java Ranger can instantiate at least one summary

Benchmark name	multi-path region summ.	+higher-order regions summ.	+single-path cases	+returns summ.
WBS	1.2E-07	1.2E-07	1.2E-07	1.2E-07
TCAS	0.24	2.5E-05	2.5E-05	2.5E-05
replace	0.63	0.90	0.12	0.12
NanoXML	1.00	1.00	1.00	0.15
PrintTokens2	0.84	0.84	0.84	0.60
ApacheCLI	0.07	0.07	0.07	0.07
MerArbiter	0.16	0.16	0.16	0.16

Table 4.4: Comparing the ratio of execution paths of Java Ranger vs SPF across the 7 benchmarks where Java Ranger can instantiate at least one summary

Benchmark name	multi-path region summ.	+ higher-order regions summ.	+ single-path cases	+returns summ.
WBS	0	0	0	0
TCAS	0.234	0	0	0
replace	0.74	1.13	0.33	0.33
NanoXML	0.9999	0.9995	0.2778	0.1332
PrintTokens2	0.92	0.92	0.92	0.69
ApacheCLI	0.05	0.05	0.05	0.05
MerArbiter	0.13	0.13	0.13	0.13

Table 4.5: Comparing the ratio of the number of solver queries of Java Ranger vs SPF across the 7 benchmarks where Java Ranger can instantiate at least one summary

RQ3: JR can be separated into the following four path-merging features.

(F1) JR only transforms multi-path regions with a single non-exceptional and non-returning exit point. This includes multi-path regions that have local, stack, field, or array outputs.

(F2) JR inlines method summaries into summaries of multi-path regions.

(F3) JR instantiates static summaries with an exceptional exit point

(F4) JR uses return value summarization to allow summaries to have a control-flow returning exit point.

To answer RQ3, we evaluated the effect each feature has as the set of features are accumulated in JR. We set up an experiment where beginning with no path-merging (aka SPF), we added path-merging features in the aforementioned order (F1-F4). For every benchmark where any path-merging was performed, we computed the ratio of a metric with a set of path-merging features enabled to the same metric's value seen without path-merging. We measured three metrics: the running time, the number of execution paths explored, and number of solver queries made. We present the results of this comparison in Tables 4.3, 4.4, 4.5,. The "multi-path region summ." column represents only enabling of the F1 feature in JR. The "+higher-order regions summ." column enables the F1 and F2 (higher-order regions) features in JR. The "+single-path cases" column enables the F1, F2, and F3 (single-path cases) features in JR. The "+returns summ." column enables all the four features with early-return summarization. Tables 4.3, 4.4, 4.5 show summarizing multi-path regions that have a single non-exceptional non-returning exit point (F1) is most often useful. This observation matches our intuition that such multi-path regions occur most frequently in Java. The addition of method summary inlining (F2) provides a major reduction in all three metrics in TCAS. This observation matches a observation made manually from TCAS' source code that multi-path regions in it often invoke methods that can be summarized by Java Ranger. The addition of single-path cases (F3) provides a major reduction in the number of solver queries in the replace and NanoXML benchmarks. Early return elimination (F4) provides a significant reduction in the number of execution paths and number of solver queries in the NanoXML and PrintTokens2 benchmarks. The benefit from this feature results from these benchmarks containing multi-path regions that contain multiple control-flow returning exit points. Tables 4.3, 4.4, 4.5 show that every path-merging feature present in JR has a beneficial impact on at least one benchmark in our set.

tool	score	cpu (s)	mem (MB)
Java Ranger	549	12000	190000
JBMC	527	25000	350000
JDart	524	22000	340000
COASTAL	472	14000	78000
SPF	410	15000	170000
Jayhorn	278	110000	530000

Table 4.6: Tool comparison on 416 SV-COMP 2020 benchmarks

4.4.5 Comparison over SV-Comp benchmarks

We also submitted Java Ranger as a participant in the International Competition on Software Verification (SV-COMP) 2020. The Java SV-COMP competition is relatively new and the 2020 benchmark suite [126] consists primarily of regression tests for the SPF and JBMC tools. In addition to the benchmarks carried over from 2019, we also submitted 13 benchmarks using the 9 subject programs used in our evaluation. We report our results in Table 4.6. Java Ranger won the Java category of SV-COMP 2020. It produced no incorrect results and 376 correct results on the 416 SV-COMP 2020 benchmarks. Java Ranger produces are "unknown" results for 40 out of 416 SV-COMP 2020 benchmarks. The breakdown of these 40 "unknown" results is as follows.

1. SPF crashes purposefully due to disabled symbolic strings. We specifically chose to make SPF crash if it finds invocation of methods like "charAt" on symbolic strings. Most of these benchmarks were from the `jbmc-regression/String*` set of benchmarks. These crashes total 22 "unknown" results.

2. Java Ranger is missing implementation of symbolic array length on multi-dimensional arrays. There are about 6 of these happening in the set of benchmarks in the `algorithm/` set. We managed to get symbolic array lengths concretized to small values in the multinewarray bytecode instruction, but ran into a bad interaction with heap path conditions maintained in SPF. 3 more symbolic array length-related crashes also result from possible out-of-bounds array accesses.

3. The third root cause of "unknown"s is timeouts. JR runs into 8 timeouts.

4. A fourth root cause is limited exploration depth. JR runs into this once with an equivalence check on ApacheCLI.

4.5 Division of Labor

This project was in joint collaboration with Soha Hussein (husse200@umn.edu). The division of labor was split along the different instantiation-time transformations described in Section 4.3.3. The primary ownership of the instantiation-time transformations can be described as follows:

1. Renaming Transformation: Soha Hussein

2. Local Variable Substitution Transformation: Soha Hussein

3. Higher-order regions transformation: Soha Hussein

4. Field References SSA form: Vaibhav Sharma

5. Array References SSA form: Vaibhav Sharma

6. Type Propagation: Vaibhav Sharma

7. Simplification of Ranger IR: Vaibhav Sharma

8. Single Path Cases: Soha Hussein

9. Translation to Green: Soha Hussein

Apart from these, the correctness-checking of region summaries and the evaluation of Java Ranger was also performed by Vaibhav Sharma.

4.6 Discussion & Future Work

Java Ranger uses semantics-preserving transformations to preserve the soundness of SSE. An example in Java where this may not be obvious is in exploration of exceptional behavior in array accesses. On encountering an array access with a symbolic index, SSE

typically explores the feasibility of an out-of-bounds array access. The array references transformation performs out-of-bounds exploration of each array access by adding a `if` statement into the summary's Ranger IR statement. This `if` statement contains a `throw` Ranger IR statement on the then-side and uses a condition that is a disjunction of the out-of-bounds conditions. Since `throw` Ranger IR statements are explored as an exceptional exit point, Java Ranger explores the possibility of the same out-of-bounds array accesses as would be explored without path-merging.

Java Ranger currently does not support recursive functions and symbolic references. Also JR does not summarize loops, it summarizes regions contained before, within, and after loops.

Java Ranger attempts to perform path merging as aggressively as possible without optimizing towards making fewer solver calls. We plan to work towards implementing heuristics that can measure the effect of path merging on the rest of the program. JR currently lacks support for symbolic object and array references. Supporting these would require integrating our implementation with SPF's lazy initialization [99] to let summaries contain symbolic object references.

Generating test cases that cover all branches is a useful application of standard symbolic execution. If applied as-is, test generation will undo the benefits of path-merging. We intend to extend JR towards test generation for merged paths in the future by targeting test generation towards a coverage criterion such as Modified Condition/Decision Coverage.

Path merging allows symbolic execution to explore interesting parts of a program sooner. But, the effect of path merging on search strategies, such as depth-first search remains to be investigated. We plan to explore the integration of such guidance heuristics with path merging in the future. Finally, we plan to expand on our formalism to prove completeness as well as soundness.

During our evaluation, we observed that Java Ranger can summarize methods in Java standard libraries. This creates potential for automatically constructing summaries of methods standard libraries so that symbolic execution can prevent path explosion originating from standard libraries.

4.7 Conclusion

We have investigated the use of static summaries to improve the performance of symbolic execution of Java. For good performance, we had to extend earlier work to account for Java's dynamic dispatch and likelihood of exceptions. Our experiment demonstrates that static summaries may yield significant performance improvements over single-path symbolic execution. Java Ranger provides evidence that inlining summaries of higher-order regions can lead to a further reduction in the number of execution paths that need to be explored with path merging as shown by the TCAS benchmark. Java Ranger reinterprets and extends path merging for symbolic execution of Java bytecode and may allow symbolic execution to scale to exploration of real-world Java programs.

Chapter 5

Using Program Synthesis For Repairing Insecure Binary Programs

5.1 Introduction

Bugs are a common and expected occurrence in software. The amount of money lost in the US in 2018 due to poor software quality was conservatively estimated at $1.1 trillion, with $479 billion of that amount lost due to effort spent on finding and fixing bugs [127]. Security bugs are a commonly-occurring category among software bugs that can have greater and more significant impact than software bugs. For example, a severe vulnerability was reported in Windows 10's X509 certificate verification that allows an attacker to impersonate websites or sign malicious code [128,129] Prevalence of security bugs has prompted the academic community to respond with sophisticated security bug detection techniques such as fuzzing. Fuzzing tools such as Driller [93], [130], and [102] are able to exploit diverse vulnerabilities such as heap and stack overflows, null dereferences, memory leaks, and out-of-bounds reads. While the bug detection research on binary code is quite successful and fast at detecting bugs, repairing these security bugs continues to remain a manual process. Security bugs are typically reported to developers and sometimes publicly disclosed later. Users of such insecure commercial

software have to wait for patches from the developers. At such times, users often wish they could identify workarounds or fixes for such issues automatically without having to wait for their software vendor to fix it. Giving users this capability would also save substantial cost savings benefit for software companies [131, 132]. However, instead of giving users the source code for commercial software, it is common for commercial software manufacturers to distribute their software only in its binary form. As opposed to source code, binary code is not meant to be human-modifiable and is optimized for execution by a computer. Repairing issues in the binary code of commercial software can be a daunting task for non-expert users since reverse engineering semantic knowledge from binary code is usually beyond the scope of their technical knowledge. Ensuring the correctness of such repairs involves not just gaining an accurate understanding of the binary code but also testing it to see if it causes other undesirable behaviors. While it is possible to expend expert human effort into repairing bugs at the binary level, this approach does not scale with a constantly increasing number of bugs, making automatic repair of binary code desirable.

A simple approach to fixing a security bug is to identify the buggy fragment of code and fix it. We present a fragment of insecure binary code in Listing 5.1.

```
0804849c <main>:
int main(int argc, char *argv[]) {
...
 char string[4];
 int i, pal = 1;
80484ad   movl   $0x1,-0x10(%ebp)
 int len = recv_delim(0, string, /* should be 4 */ 32, '\n');
80484b4:  push   $0xa
80484b6:  push   $0x20 ; should be 4
80484b8:  lea    -0x1c(%ebp),%eax
80484bb:  push   %eax
80484bc:  push   $0x0
80484be:  call   804843b <recv_delim>
...
```

Listing 5.1: Fragment of binary code with interleaved source code that makes a call to the `recv_delim` method presented in Listing B.1

Listing 5.1 shows the disassembly of a fragment of the `main` method which calls the `recv_delim` method with the address of its local 4-byte array, `string`, passed as an argument. Even though `string` is only 4 bytes long, `main` erroneously passes 32 as the third argument to `recv_delim`. The source code for `recv_delim` is presented for reference in Listing B.1 in Appendix B. Since the third argument to `recv_delim` method is 32, `recv_delim` reads up to 32 bytes of input from the standard input as long as the newline character is not provided as an input byte. This incorrect parameter value of 32 is a security bug because it allows an attacker to not only overwrite the local variables, `i`, `pal`, `len`, but also overwrite the return address of the `main` method. Overwriting the return address of an executed method with an attacker-controlled value is a useful primitive for constructing a successful attack against a program. The change required to fix the security bug is simple, the third argument that should be passed to `recv_delim` should be a constant value that prevents the return address from being overwritten. In fact, the most desirable change is one that not only fixes the security bug but also preserves the functional correctness of the rest of the `main` method. In case of the repair on the source line on line number 7 of Listing 5.1, this most desirable change is to pass 4 instead of 32 as the third argument to `recv_delim`. This constant value of 4 would prevent both the local variables in `main` and its return address from being overwritten, thus preserving the functional correctness of `main`.

Finding such a functionally correct repair for a security bug automatically presents several challenges. First, the security conditions have to be constructed that detect out-of-bounds writes to the `string` variable inside `recv_delim`. Second, for all writes that lie within the bounds of `string`, the correct constant value of 4 has to be inferred and passed as the third parameter to the `recv_delim` method. Automatically constructing this repair that preserves functional correctness of this fragment of binary code in `main` requires both of these steps to be performed correctly.

In this chapter, we present a technique, when given a set of properties that detect any insecure behavior in fragment of code, can construct a repair that fixes violation of the security properties while preserving the functional correctness of the code fragment. We refer to the fragment of code with such security property violations as the *target* code. This terminology bears similarity to uses of this term in‘chapter 2. Our program repair approach extends the CEGIS algorithm defined in chapter 1 to ensure that our

repairs do not violate any security property for all inputs to the target code. We use symbolic execution for implementing each individual step in the CEGIS loop. We heuristically select small target fragments to avoid path explosion during the repair verification step. We present an evaluation of our implementation by automatically constructing the functionality-preserving repair for four binary programs used in the DARPA Cyber Grand Challenge (CGC). We find that our implementation can find the most desirable repair for security bugs in these four programs in a reasonable time budget.

In the rest of this chapter, section 5.2 presents related work, section 5.3 presents our extension of CEGIS, and section 5.4 describes our implementation of our technique. Next, section 5.5 presents an evaluation of our implementation on four DARPA CGC programs. Finally, section 5.6 presents a discussion of further extensions to our approach, and section 5.7 concludes this chapter.

5.2 Related Work

To put our proposed work in context, we describe approaches that are complementary to our approach for program repair, and then compare our approach to other program repair approaches. We do not attempt to provide a complete survey; we refer interested readers to existing survey papers mentioned in each area of research below.

5.2.1 Security bug detection

Our program repair approach depends on a set of security properties being provided by a bug detection tool. These properties can take the form of multiple related software security bugs found by a bug detection tool. Fuzzing is one of the most popular and easiest-to-deploy bug-finding techniques. The central idea of fuzzing is to run a program with a stream of random inputs and detect violations of a correctness policy. The most recent survey on fuzzing by Manès et al. [133] breaks fuzzing down into three broad categories. Fuzzing, when only allowed to provide inputs to the program-under-test and monitor its outputs, is termed as black-box fuzzing. This is in contrast with white-box fuzzing techniques [113] which uses knowledge of the internals of the program-under-test to generate inputs. A third category of fuzzers is grey-box fuzzing. This category spans

tools such as AFL [134] which uses imprecise information about the program-under-test to be fast at generating inputs. A combination of these fuzzing techniques was employed by teams participating in the DARPA CGC. In particular, the Mechanical Phish team built a system named Driller [93] which used a combination of grey-box and white-box fuzzing. Driller used AFL to find bugs and angr [94], a dynamic symbolic execution tool, to construct inputs to satisfy difficult, specific checks that could only be satisfied by a small set of inputs. Another aspect of the DARPA CGC was patching security bugs so that they cannot be exploited by other teams. The following section describes techniques that may be used for this purpose.

5.2.2 Failure-oblivious Computing and Fault Tolerance

Failure-oblivious computing [135] is an area of research that attempts to allow program execution to continue through memory errors without memory corruption. It allows program execution to continue by manufacturing values for illegal reads and ignoring illegal writes. This work has been further extended by runtime tools [136], [137] that attempt to make programs fault-tolerant to memory errors by using randomization and replication while probabilistically maintaining soundness. While these approaches reduce the exploitability of insecure implementations, they do not provide any guarantees about not affecting the functional correctness of the implementation. Our approach preserves the functional correctness while repairing only the insecure behavior in the target fragment. However, since our approach is limited to repairs within a fixed set of candidates, techniques from failure-oblivious computing are complementary to our approach.

5.2.3 Binary Hardening

Another approach to defend against exploitation of memory errors is to harden the binary and make the attack more challenging. This approach was taken by the Zipr [138] tool that rewrites a binary to augment it with control-flow integrity when it was used in the DARPA Cyber Grand Challenge competition (CGC).

Similar techniques were also adopted by the Mechanical Phish team in the DARPA

CGC competition [139]. Specifically, the Mechanical Phish team built a binary patching system named *Patcherex*. Patcherex attempted to modify binaries by using generic binary hardening techniques without using knowledge of the vulnerabilties in the binary. It made use of three kinds of hardening techniques. First, they added checks to loosely enforce control-flow integrity and prevent memory-leaking exploits. Second, they added techniques to make analysis of their patched binaries difficult for other CGC participants. An example of this is their use of bugs in QEMU, which was used by other teams. Third, they applied common binary optimization techniques such as constant propagation and dead assignment elimination.

Other techniques that can be applied to achieve binary hardening are stack randomization [140], dynamic canary randomization [141], dynamic code mixing [142]. These techniques were also combined as part of larger systems such as Helix [143] to strengthen binaries against attacks. Such systems use such hardening techniques to make the attack more challenging and fall back to using traditional program repair tools (such as GenProg [144] for Helix) when the security bug can still be exploited. Another related application of binary hardening techniques is to create multiple variants of a binary using different hardening transformations and run them in parallel. Increasing the diversity between variants reduces the probability that an attack mounted through the same input stream will be successful across all variants. This approach was used by tools such as Double Helix [145] to achieve security through diversity which is also an active area of research [146]. These approaches are complementary to our proposed approach because these techniques can be applied in situations where a repair within our grammar cannot be found.

5.2.4 Reassembleable Disassembly

Reassembleable disassembly was an approach adopted by the Mechanical Phish team in the DARPA CGC to rewrite binaries so that they became harder to exploit by other teams. A reassembleable disassembly tool extracts relocatable assembly from the binary without the use of debugging or relocation information. Later, it assembles the assembly code back into an executable binary. Uroboros [147], [148] was one of the the first end-to-end systems to provide such functionality. The primary challenge in making a disassembly relocatable lies in determining whether an immediate, used as an

operand within an assembly instruction, is used as an integer or as a label referencing another location. The process of identifying such references in the disassembly is called *symbolization*. Uroboros performs its symbolization by using an assumption that the original binary would not make invalid memory accesses. It filters out any immediate operands which are outside the address space allocated for the binary. However, Uroboros still fails to rewrite many binaries and the authors of Uroboros had to make manual adjustments for some binaries.

Ramblr [149] (built on top of angr [94]) uses additional analyses to its symbolization. It is also the extension of the Patcherex system used by the Mechanical Phish team in the DARPA CGC. Ramblr reduces the number of candidate immediates for symbolization via local data flow and value set analysis. It identifies data in code and performs type inference on immediate operands. It also infers array bounds to identify more complex data structures and avoid symbolizing such locations. These analyses allow Ramblr to account for more compiler optimizations or value collisions.

Another approach to reliable disassembly is the superset disassembly approach of Multiverse [150]. Multiverse avoids any assumption about the usage of memory addresses or immediates. Instead, it performs disassembly at every byte offset of the text section. Linear disassembly from every byte offset is stopped whenever it reaches a previously visited byte offset or an invalid instruction. Thus, Multiverse obtains a superset of the correct disassembly. To allow reassembly from this superset, a mapping from an address in original binary into the new binary is maintained. This mapping is looked-up at runtime to resolve any indirect control transfers in the new binary. A cost for the reliability provided by Multiverse is a larger code size expansion compared to other static rewriting approaches.

5.2.5 Adapter Synthesis

Our proposed approach to binary program repair builds on existing research described in Chapter 2. Adapter synthesis can be applied as-is for applications such as reverse engineering and deobufscation where the goal is to find a reference implementation that performs the same functionality. But, for applications such as program repair, adapter synthesis needs to be extended to prevent undesirable behavior in the target implementation from appearing in the adapted reference. This extension is along two

directions.

1. We need to extend adapter synthesis to perform adaptation modulo an undesirable behavior.

2. While adapter synthesis finds an adaptable substitution for all possible inputs to the target implementation, program repair needs to only repair uses of the target implementation. In other words, program repair needs to take the context of usage of the target implementation into account.

We perform these same extensions for our program repair technique too. But, we do not need to find a reference implementation to adaptably repair the buggy target. Not requiring a reference implementation is restrictive for our approach since it has to synthesize the entire repair in the buggy target code. However, not requiring a reference implementation helps our approach along one dimension of scalability: we do not need to run the synthesis on every combination buggy target code and reference implementation.

5.2.6 Program Synthesis

Our proposed approach fits within the syntax-guided synthesis domain described in Section 2.6.5. We provide our technique a fixed set of repair candidates described by a context-free grammar. Our repair technique then attempts to find a repair within the given set of candidates. Our repair technique also uses an extension of the CEGIS algorithm described in subsection 1.2.1 However, our use of program synthesis extends the CEGIS algorithm in two ways.

1. We break the verification step down into two independent checks. The first one verifies the repair for functional correctness and the second one verifies it for meeting security properties. Failure of these two independent verification steps produce two different sets of test inputs.

2. Our synthesis step searches for a repaired target fragment that satisfies both functional correctness and security properties represented by the two sets of test inputs.

Several approaches to using synthesis for program repair have already been attempted which we describe in the following section.

5.2.7 Program Repair

The area of automated program repair has seen a tremendous interest in the research community in the last few years. A comprehensive survey of this research area was recently presented by Gazzola et al. [151]. The closest area of research for generating a fix automatically comes from the set of tools that use a *generate-and-validate* approach. These tools [144], [152], [153], [154] use an iterative approach that is similar to adapter synthesis. They first generate a candidate fix to a repair problem in the generate step. Next, they validate the correctness of the candidate fix in a validate step by running a set of test cases. The correct fix should pass all available tests. We share the similarity of synthesizing a repair against a small set of tests. However, we validate our repairs by checking it for functional correctness and non-violation of security properties against all inputs to the target fragment. Code Phage [152] attempts to fix bugs in binary code by automatically transferring code from a reference (referred to as *donor*) into a target (referred to as *recipient*) implementation. Another tool which shares intuition with both Code Phage and adapter synthesis in proposing the use of existing reference implementations for program repair is SCRepair [153]. SCRepair makes use of metrics that establish the similarity and differences between code fragments. Their metrics can be reinterpreted for binary code and used for ranking candidate repairs to address scalability challenges involved with a large set of program repair candidates.

The second approach, which is much closer to our approach, to automated program repair is semantics-driven repair. This approach makes use of a formal specification of a repair problem and uses it to find a solution that is guaranteed to solve the repair problem. A number of semantics-driven repair tools [155], [156] make use of oracle-guided program synthesis. Our proposed approach shares a similar intuition with these tools and makes use of a target implementation as the oracle. More specifically, our proposed approach shares the intuition with Angelix [157] that symbolic execution can be used to derive semantic information from a target implementation. Our approach also shares with Angelix the creation of an angelic forest of repair candidates. However, a key difference between our approach and Angelix, SemFix, and DirectFix is that our repairs use the functional correctness of the buggy target implementation as the specification that has to be met. Also, our repairs are targeted towards repairing violations of security properties as opposed to repairing functional incorrectness. Finally, the implementation

of our approach operates on binary code, which is a significant departure from other program repair tools. Operating at the binary level presents a unique set of challenges such as resolving indirect jumps and access into symbolic regions of memory [158] Another reference implementation-based program repair technique is proposed in a tool named SOSRepair [159]. Afzal et al. perform source-level repairs using semantically equivalent snippets of code that come from the same project as the buggy code region. They use tests to encode a profile of buggy and non-buggy behaviors of the buggy code region and search for candidate snippets that match this profile. Afzal et al. report that snippets that are too small (1-3 LoC) will usually overfit the tests whereas snippets that are too large (7-9 LoC) will have less likelihood of having redundancy in the same project. Their evaluation shows that they can repair more bugs in larger source code repositories. We share the intuition with Afzal et al. that repairs should avoid overfitting to a fixed set of tests.

Several benchmarks [160], [161], [162], [163] have also been developed used for comparing different automated program repair tools. While some of these (such as the ManyBugs and IntroClass [163]) can be compiled to binary code and used to evaluate binary program repair tools, the evaluation would be susceptible to compiler optimizations and choice of compiler. Since our repairs are targeted towards repairing violations of security properties, we adopt the DARPA CGC benchmark set for our evaluation.

5.3 Technique

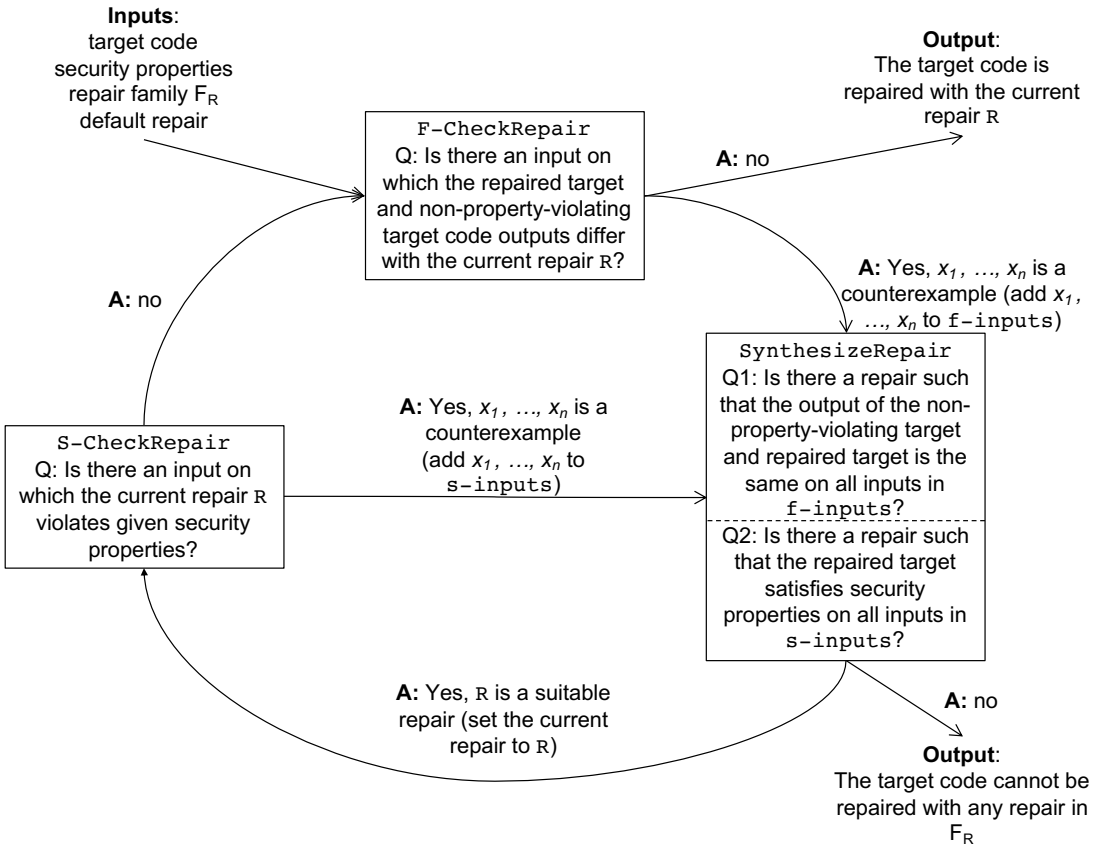


Figure 5.1: Counterexample-guided repair synthesis to satisfy a set of security properties in a buggy target code

We present a high-level overview of our technique in Figure 1.1. Our technique extends the CEGIS loop presented in Figure 1.1 by breaking down the correctness verification step into verification of two independent correctness properties. The first correctness property is that the repaired target code should be functionally as correct as the original target code when it did not violate any security property. The second correctness property is that the repaired target code should not violate any security property for all inputs to it. We explain our algorithm for synthesizing repairs in the following section.

5.3.1 Algorithm

The synthesis procedure begins with inputs that specify the target code that is to be repaired. The target code can be of any granularity, a function, or an arbitrary fragment. Our implementation considered the target code to be an arbitrary fragment of code. The second input is a set of security properties that the target code should satisfy. We assume these properties to come from a separate tool that detected the insecure behavior in the target code. The third input to our approach is a set of candidate repairs as described by a family of repairs, F_R. The fourth input is a default repair that is chosen arbitrarily from the set of repairs described by F_R. This default repair is set to the current repair, `R`, for the repair synthesis process to begin.

The algorithm begins by running `F-CheckRepair`, where `F` represents checking the repair for functional correctness. `F-CheckRepair` checks if the current repair, `R`, provides the same functional correctness as provided by the target code when it does not violate any security property. This step is similar to the `CheckAdapter` step in Figure 2.1 in chapter 2. If `F-CheckRepair` can find an input that causes the repaired target's output to be different from the target code's output without violating any security property, then such an input is proof that the repaired target code is functionally incorrect. This input is then added to a list of tests, `f-inputs`, representing violation of functional correctness by previous repair candidates.

After `F-CheckRepair`, the algorithm proceeds to the `SynthesizeRepair` step. `SynthesizeRepair` is similar to the `SynthesizeAdapter` step shown in Figure 2.1. But, instead of finding an adapter that is only functionally correct, the `SynthesizeRepair` step has two independent goals. `SynthesizeRepair` needs to find a repair that is not only functionally correct but also satisfies the provided security properties. These two goals are represented by the two questions, `Q1` and `Q2`, in Figure 5.1. The first goal is satisfied when the repaired target code's output matches the output of the target code without violating any security property for all inputs in the `f-inputs` list of inputs. The second goal is satisfied when the repaired target code can be run for all the inputs in the `s-inputs` list without violating any security property. If no repair in the set of candidate repairs can satisfy both goals, represented by inputs in `f-inputs` and `s-inputs`, then the procedure stops with the conclusion that no repair exists. Otherwise, the candidate repair `R`, which satisifies both goals, is passed to the `S-CheckRepair` step.

The `S-CheckRepair` step runs the target code, with the current repair `R` applied to it, It searches for inputs that would cause the repaired target code to violate the security properties. If any such inputs are found, they are added to the `s-inputs` list, and the `SynthesizeRepair` step is run next. If no such inputs can be found, then the current repair `R` does not violate any security properties and only needs to be checked for functional correctness. This check for functional correctness is run next by executing the `F-CheckRepair` step.

5.3.2 Specifying security properties

The algorithm presented in Figure 5.1 depends on a set of security properties being provided as input. These security properties are derived from the insecure behavior that was detected by the bug-reporting tool. Sophisticated bug-reporting tools such as Mayhem [102], Driller [93], and Qsym [130] can find violations of implicit security properties. Examples of these security properties include checking for out-of-bounds memory access, return address overwrites, and use-after-free errors. We provided an example of a fragment of code in Listing 5.1. The writing to memory beyond the length of `string` defined in `main` happens in the `recv_delim` method defined in Listing B.1.

If an input read in the `recv_delim` method happens to be long enough that it can even overwrite the return address stored on the stack. We explain the offset of the return address from the beginning of the `string` stack buffer in Figure.

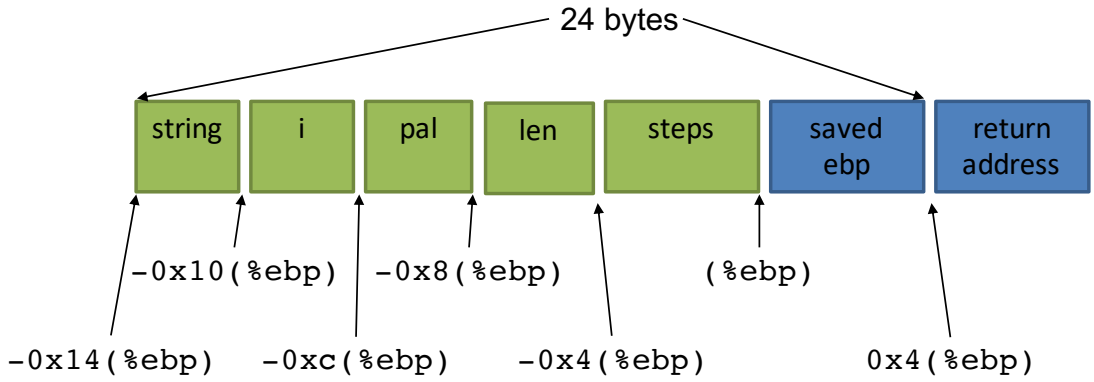


Figure 5.2: Stack layout of the `main` method stack frame which allows a buffer overflow via a call to `recv_delim`

We wish to remind readers that the `string` local stack buffer is declared as being 4 bytes long in Listing 5.1. The return address of the `main` stack frame is at a 24 byte offset from the beginning of the `string` stack buffer. If an input from the user manages to be at least 24 bytes long, it can overwrite the return address, thereby causing instruction execution to return to a code address of the attacker's choosing, after execution through `main` has finished. Security bug detection techniques can detect such return address overwrite operations and stop the program execution from returning to an attacker-controlled code address. Thus, preventing an overwrite of the return address with an attacker-controlled value is a security property. However, just enforcing this security property is not enough for the bug to be repaired. Inputs of other lengths can still be constructed that do not overwrite the return address but overwrite the values of other local variables, `i`, `pal`, `len`, and `steps`. Allowing the values of these local variables to be modified via writes to the `string` variable affects the functional correctness of the target code without violating the security property that the return address of a method should not be overwritten by an attacker-controlled input. Therefore, our program repair approach depends on the bug detection tool being able to also detect store operations to these local variables that are performed through the base pointer to `string`. Our approach relies on being provided such additional security properties that ensure that store operations to `string` do not modify any memory that lies outside the bounds of the `string` buffer. We consider construction of such security properties out of scope for this chapter. Techniques for constructing such properties based on a crash or a security bug is an active area of research. Automated techniques for diagnosing a crash using backward taint analysis [164], [165], [166] as well as value set analysis for binary code [167] can be used to construct such security properties.

5.3.3 Functional correctness checking

Checking the security properties helps the technique find a repair that fixes insecure behavior in the target code, as verified in the `S-CheckRepair` step. Another crucial correctness property that must be satisfied by the repair is functional correctness which is verified in the `F-CheckRepair` step. In order to ensure this property, the `S-CheckRepair` uses the same test harness as described in Listing 2.7. The `F-CheckRepair` needs to compare outputs of the non-security-property-violating target cod with the outputs of

the repaired target code. This comparison is performed by capturing all side-effects of the target and the repair via system calls and writes to memory and comparing them for equality using a Satisfiability Modulo Theories (SMT) solver. The use of an SMT solver ensures that the side-effects are compared for not just syntactic but also semantic equality.

5.3.4 Target code selection

Having a complete set of security properties over the target code is crucial for our program repair tool to find the right repair. A different aspect of our setup that affects the performance of the tool is the number of control-flow paths through the target code. If the target code contains an infinitely large number of paths, determining the repaired target's functional equivalence with respect to the target code is undecidable. Therefore, it is important to break the target code into fragments against which the repaired target can be checked for functional correctness within a reasonable time budget. In order to select target fragments, we used a target fragment selection heuristic which can be explained by means of two examples presented in Figure 5.3 and Figure 5.4.

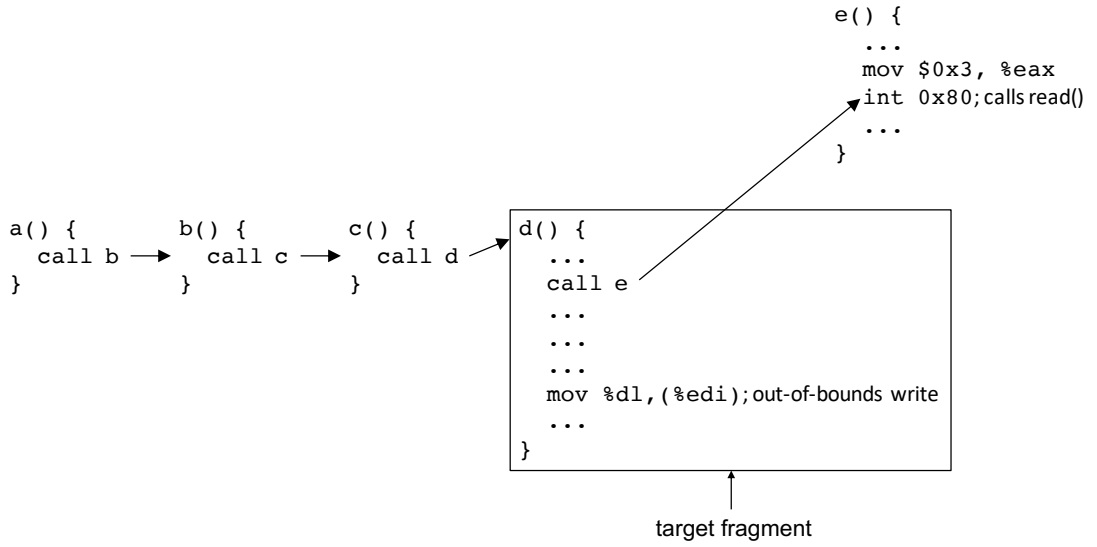


Figure 5.3: Target fragment selection for a part of a call-graph where the direct or indirect caller of the last input read operation is in the same stack frame as the last instruction that violates a security property

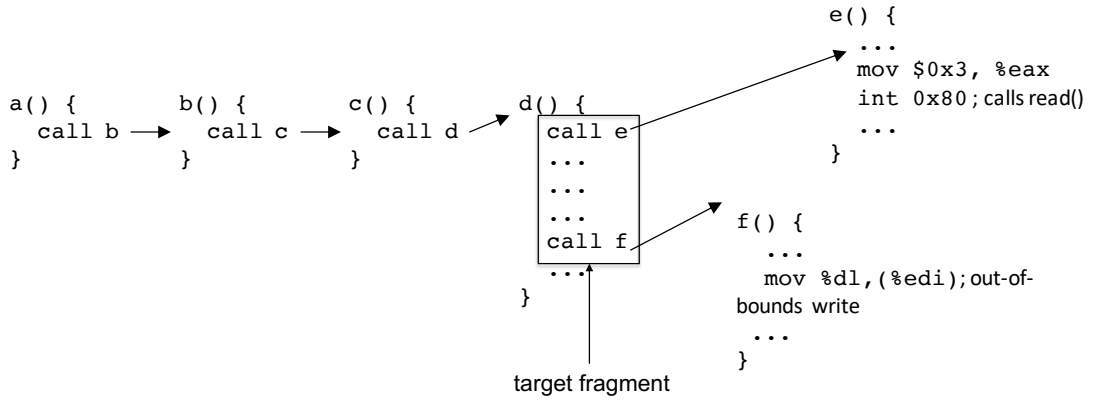


Figure 5.4: Target fragment selection for a part of a call-graph where the direct or indirect caller of the last input read operation is in a different stack frame than the one where a security property violation occurs

Figure 5.3 shows a partial interprocedural call-graph where the method `d` calls method `e` which reads in inputs to the program via te `read` system call. In this case, the

last instruction to violate a security property lies within the stack frame of `d` itself. This case is distinguished from the case where this last security property violation may lie in another callee of `d` as shown in Figure 5.4. We wish the target fragment to span all the security property violations, thereby requiring our target fragment to end only after the last security property violation has been reported. But we would also like our target fragment to be as small as possible. Therefore, we make a heuristic choice to begin the target fragment right before the last input to the program is read. Therefore, Figure 5.3 chooses the target fragment to be the entire method `d`. In contrast, Figure 5.4 chooses the target fragment to begin when arguments to `e` are construct and end after the call to `f` has completed. The selection of the target fragment does not need to be accurate as long as it captures all the security violations and allows for at least one input to be read.

5.4 Implementation

We implemented the counterexample-guided repair synthesis described in section 5.3. For the `F-CheckRepair`, `SynthesizeRepair`, and `S-CheckRepair` steps, we implemented the search procedure using FuzzBALL [21], a binary symbolic execution engine. We implemented the test harness described in Listing 2.7 for X86 binary code. All the three steps of our technique were given instruction addresses for the beginning and end of the target fragment. To run the test harness, the `F-CheckRepair` step effectively ran the target fragment twice, first with the repair and then without the repair. The `S-CheckRepair` step ran the target fragment once, with the repair applied. Finally, the `SynthesizeRepair` step ran the target fragment twice for every input in `f-inputs`, and later once for every input in `s-inputs`.

For the repair family, we re-used the argument substitution adapter family described in subsection 2.2.2. In addition to this family, we added a repair symbolic variable in the `SynthesizeRepair` step to locate the instruction address where a repair should be applied. This addition avoids the need for fault localization and instead includes localization of the repair in the repair synthesis process. While this addition also contributes a multiplicative factor in the size of the repair search space, our repair synthesis process is unaffected by it because of its use of symbolic execution.

5.5 Evaluation

We evaluated our program repair approach on four case studies chosen from a set of software programs used for the qualifying round of the DARPA CyberGrand Challenge (CGC). The DARPA CGC [168] qualifying round consisted of a set of binaries with built-in vulnerabilities. Participating teams had to provide automated tools to exploit these vulnerabilities. The tools could also release their own patched versions of the binaries which could be attacked by other teams. The binaries used in the DARPA CGC were built for the Decree Operating System, a simplified operating system built by DARPA just for the CGC. Since no real-world binaries operate on the DECREE operating system and our repair implementation supports X86 binary code, we used the cb-multios port of the CGC programs [169]. We describe the setup for our four case studies in the following section.

5.5.1 Setup

For every program in the cb-mulitos collection, the collection contains source code and scripts to build it. It also has inputs that will trigger the known vulnerability in each program as well as the patch that should fix the vulnerability. The vulnerability-triggering input is referred to as PoV, an abbreviation for Proof of Vulnerability, for the rest of this chapter. We first built four binaries for our case studies using the build scripts provided in cb-multios. The four programs from the DARPA cGC benchmark set used in this evaluation are Palindrome, WhackJack, The_Longest_Road, and ShoutCTF. We will use TLR to refer to The_Longest_Road for the rest of this chapter. Next, we confirmed the presence of a vulnerability in each of these programs by running each program with its PoV under FuzzBALL. FuzzBALL was able to detect exploitability or a crash for all four of these case studies. Finally, we used the bug detection or crash triggered by the PoV on each binary to construct security properties to be provided to our repair tool.

The last remaining input to our program repair approach is the target fragment. We collected stack backtraces from running the PoV on each binary and ran our heuristic for target fragment selection. This gave us four target fragments, one on each case study, to find repairs in.

5.5.2 Results

case study	# sym. inputs	size of repair space	total time (min)	steps (F-CR, SR,S-CR)	last S-CR time (min)	last F-CR time (min)
Palindrome-8sym-inputs	8	321991304	11.85	(3, 3, 3)	1.07	3.67
Palindrome-pm-64sym-inputs	64	321991304	46.25	(3, 3, 3)	2.45	32.08
WhackJack	4	2253939128	2.34	(4, 4, 4)	0.5	0.18
TLR	5	223368	21.7	(3, 3, 3)	1.34	0.99
ShoutCTF	128	28262158128	82.2	(2, 1, 1)	23.27	14.48

Table 5.1: Summary of our automated program repair results on four DARPA CGC case studies along. The Palindrome case study is shown twice because we ran program repair on two versions of it. The second version was compiled with the `-O1` compiler flag which enabled conditional move instructions.F-CR, S-CR, and SR refer to the `F-CheckRepair`, `S-CheckRepair`, and `SynthesizeRepair` steps respectively.

With all the inputs set up for our program repair tool, we ran our automated program repair implementation on the four case studies and present our results in Table 5.1. The first and second columns report the case study and number of symbolic inputs used to verify the repair in the case study respectively. The third column reports the size of the repair space. This size was computed using a argument substitution repair family formula similar to the one presented in section 2.4.7. We multiplied the number of available repair locations with the size reported for this family to get the total size of repair space reported in the third column. The fourth column reports the total time taken to find the right repair. The fifth column reports the number of steps executed to find the right repair. The sixth and seventh columns report the time spent in executing the last `S-CheckRepair` and `F-CheckRepair` steps respectively.

For the first case study, Palindrome, we found that the number of execution paths during the `F-CheckRepair` and `S-CheckRepair` steps was exponential in the number of symbolic inputs. Since Palindrome takes 64 input bytes, this led to each verification steps running for several days before finding a counterexample. On manually examining the symbolic branches within the selected target fragment in Palindrome, we found that these side-effects of these symbolic branches could easily be summarized using veritesting. However, FuzzBALL does not have veritesting support. To simulate the presence of veritesting, we recompiled the Palindrome binary with the `-O1` gcc compiler optimization flag. Using this flag, fortunately, caused gcc to use conditional move instructions to set the value of a local variable in a method in Palindrome. Since FuzzBALL generates Vine IR if-then-else expressions to execute conditional move instructions, this compiler optimization turned the number of execution paths explored by FuzzBALL during the repair verification steps from exponential to linear in the number of symbolic inputs. Therefore, we present results of these two versions of Palindrome in the first two rows of Table 5.1.

The first row reports results from verifying Palindrome's repairs using 8 symbolic inputs, the first four of which were the prefix and the last four were the suffix in the list of 64 input bytes. The second row reports results from verifying Palindrome's repairs with all 64 input bytes set to unconstrained symbolic variables. Both rows show that the last `F-CheckRepair` and `S-CheckRepair` took a significant portion of the total synthesis time. The third and fourth rows in Table 5.1 shows that the size of the repair space does not affect the total time needed to find the right repair. The size of the adapter search space is small in the fourth row because we ran this repair with a smaller constant bound than the other repairs. However, these bounds were chosen arbitrarily and can easily be fixed to the same bound for all of our case studies. The fifth row shows that the repair for ShoutCTF took significantly longer than repairs for the other three case studies. However, a significant portion of this time was spent in running the `F-CheckRepair` and `S-CheckRepair` steps. These steps are significantly slower for ShoutCTF because every input byte read from standard input has to be translated via lookup in a table with 256 entries. We were able to make these lookups happen much faster by using FuzzBALL's support for the theory of arrays with the SMT solver. However, every lookup still negatively impacted the total time required to verify repairs in ShoutCTF.

5.6 Discussion

As shown in the previous section, our program repair technique finds the right repair for four DARPA CGC programs in a reasonable time budget. The technique can be improved along a few dimensions. First, more repair families can be implemented and evaluated against more CGC programs. Synthesizing the correct repair is unlikely to scale to large repairs in binary code. For example, we observed that synthesizing an arithmetic adapter involving only two operands caused more timeouts in section 2.4.7. This points to an intuition that our synthesis tool should attempt to synthesize smaller repairs first. On manually examining patches to DARPA CGC programs provided as part of the cb-multios [169] collection of programs, we found several programs only need small changes to the branching condition used in a if-statement to be correctly repaired. Such repair grammars should be added to our repair tool to allow a diversity of repairs.

Our repair tool uses symbolic execution for both the synthesis and verification steps. Even though we attempted heuristically choosing small target fragments to repair first, the Palindrome case study demonstrated the need for path-merging to avoid having to explore an exponential number of paths. Having path-merging would allow our program repair tool to be faster at finding and verifying repairs.

Our program repair approach is concerned with finding the correct repair. However, it would be most useful to users of commercial software if this repair can also be applied to the insecure binary to produce a patched binary. This extension requires using binary rewriting tools such as DynInst [170] or reassembleable disassembly tools such as Ramblr [149] or Uroboros [147] to patch the repair into the original binary. However, these tools also have scalability challenges of their own as reported by Emamdoost et al [171].

5.7 Conclusion

We presented an automated program repair technique that extends the CEGIS algorithm for finding repairs in binary code. Given a set of security properties in a target code, our approach finds repairs that fix violation of the security properties while preserving the target code's functional correctness. We presented a heuristic for identifying small target fragments that make the repair verification task tractable. Our program

repair tool finds the correct repair for four binary programs used in the DARPA CGC in a reasonable time budget. Our results suggest that the application of program synthesis for binary program repair is promising and can be applied to automate the process of repairing binary code.

Chapter 6

Conclusion

In this dissertation, we presented the thesis that the scale of functionality, that can be automatically found to fit a specification, is significantly improved by making use of a reference implementation. While a CEGIS loop is commonly used to automate such a search process, we found bounded symbolic execution can be used for both steps of the CEGIS loop. Automatically synthesizing an adapter to match the interfaces of a target and reference implementation enabled several applications such as reverse engineering, deobfuscation, library replacement, and even program repair. While adapter synthesis is a useful technique, allowing users of the technique the flexibility to specify their own adaptation operators in a source-level language is desirable. We presented path-merging as a method to allow flexibility in the adapter specification. We found that our extension of previous approaches to path-merging for Java bytecode provided significant scalability benefits to symbolic execution. Finally, we found adapter-like operations such as argument substitution can serve to repair security bugs in binary code, even without the use of a reference implementation. Our program repair approach preserves functional correctness in target code while repairing the security bug present in it.

References

[1] J. Bornholt. Program Synthesis Explained. https://www.cs.utexas.edu/~bornholt/post/synthesis-explained.html, January 2015.

[2] J. Bornholt. Program Synthesis in 2019. https://blog.sigplan.org/2019/07/31/program-synthesis-in-2019/, July 2019.

[3] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, Oct 2013.

[4] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *36th International Conference on Software Engineering (ICSE)*, pages 1083–1094, June 2014.

[5] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 193–204, June 2012.

[6] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser. Java Ranger for SV-COMP (to appear). In *SV-COMP Presentations at the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020.

[7] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser. Java Ranger, December 2019.

[8] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. In *Proceedings*

of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, page 287–296, New York, NY, USA, 2013. Association for Computing Machinery.

[9] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[10] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, October 2006.

[11] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, New York, NY, USA, 2013. Association for Computing Machinery.

[12] S. Saha, P. Garg, and P. Madhusudan. Alchemist: Learning guarded affine functions. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, pages 440–446. Springer International Publishing, 2015.

[13] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[14] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[15] V. Sharma, K. Hietala, and S. McCamant. Finding Substitutable Binary Code By Synthesizing Adaptors. In *11th IEEE Conference on Software Testing, Validation and Verification (ICST)*, April 2018.

[16] V. Sharma, K. Hietala, and S. McCamant. Finding substitutable binary code by synthesizing adapters. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[17] V. Sharma, K. Hietala, and S. McCamant. Finding substitutable binary code for reverse engineering by synthesizing adapters. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 150–160, April 2018.

[18] H. Borck, M. Boddy, I. J. D. Silva, S. Harp, K. Hoyme, S. Johnston, A. Schwerdfeger, and M. Southern. Frankencode: Creating diverse programs using code clones. In *2016 IEEE 23nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 604–608. IEEE, 2016.

[19] VLC - Official Site - VideoLAN. `https://www.videolan.org/vlc/index.html`, 2017.

[20] K. Hietala. Detecting Behaviorally Equivalent Functions via Symbolic Execution . Undergraduate Honors thesis, University of Minnesota – Twin Cities, 2016. Retrieved from the University of Minnesota Digital Conservancy, `http://hdl.handle.net/11299/181385`.

[21] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 337–348, New York, NY, USA, 2012. ACM.

[22] FuzzBALL: Vine-based Binary Symbolic Execution. `https://github.com/bitblaze-fuzzball/fuzzball`, 2013–2016.

[23] D. Song. Bitblaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`, 2017.

[24] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[25] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*, pages 519–531, Berlin, Heidelberg, 2007. Springer.

[26] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System V Application Binary Interface. `http://www.x86-64.org/documentation/abi.pdf`, 2013.

[28] FireEye Inc. Shifu Malware Analyzed: Behavior, Capabilities and Communications. `https://www.fireeye.com/blog/threat-research/2015/10/shifu-malware-analyzed-behavior-capabilities-and.html`, 2015.

[29] Shifu: 'Masterful' New Banking Trojan Is Attacking 14 Japanese Banks. `https://securityintelligence.com/shifu-masterful-new-banking-trojan-is-attacking-14-japanese-banks/`, 2015.

[30] Palo Alto Networks. 2016 Updates to Shifu Banking Trojan. `http://researchcenter.paloaltonetworks.com/2017/01/unit42-2016-updates-shifu-banking-trojan/`, 2017.

[31] H. S. Warren. Collection of programs to compute CRC-32 checksum. `http://www.hackersdelight.org/hdcodetxt/crc.c.txt`, 2013.

[32] H. S. Warren. *Hacker's delight*. Pearson Education, 2013.

[33] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, New York, NY, USA, 2012. ACM.

[34] T. A. Proebsting. Optimizing an ansi c interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 322–332, New York, NY, USA, 1995. ACM.

[35] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[36] T. László and Á. Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[37] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[38] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[39] Implementing Strassen's Algorithm For Matrix-Matrix Multiplication With OpenMP-3.0 Tasks (Intel). `https://software.intel.com/en-us/courseware/256200`, 2010.

[40] DARPA. Space/Time Analysis for Cybersecurity (STAC). `http://www.darpa.mil/program/space-time-analysis-for-cybersecurity`, 2015.

[41] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, pages 463–473. IEEE Computer Society, 2009.

[42] caml.inria.fr. Module hashtbl. `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Hashtbl.html`, 2017.

[43] cppreference.com. std::map - cppreference.com. `http://en.cppreference.com/w/cpp/container/map`, 2017.

[44] GNU. The GNU C Library : Absolute Value. https://www.gnu.org/software/libc/manual/html_node/Absolute-Value.html, 2017.

[45] EC2 Instance Pricing – Amazon Web Services. https://aws.amazon.com/ec2/pricing/on-demand/, 2018.

[46] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[47] Rockbox - Free Music Player Firmware. https://www.rockbox.org/, 2017.

[48] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, 2011. ACM.

[49] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[50] B. Dutertre. Yices 2.2. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 737–744, Cham, 2014. Springer International Publishing.

[51] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 223–234, June 2009.

[52] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[53] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[54] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[55] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, New York, NY, USA, 2009. ACM.

[56] P. Amidon, E. Davis, S. Sidiroglou-Douskos, and M. Rinard. Program fracture and recombination for efficient automatic code reuse. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6, Sept 2015.

[57] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 266–280, New York, NY, USA, 2016. ACM.

[58] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella. Search-based synthesis of equivalent method sequences. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 366–376. ACM, 2014.

[59] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik. Template-based circuit understanding. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 17:83–17:90, Austin, TX, 2014. FMCAD Inc.

[60] D. Katz, J. Ruchti, and E. Schulte. Using recurrent neural networks for decompilation. In *Software Analysis, Evolution and Reengineering (SANER), 2018*. IEEE, 2018.

[61] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 276–291, Washington, DC, USA, 2014. IEEE Computer Society.

[62] M. Rittri. Using types as search keys in function libraries. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 174–183, New York, NY, USA, 1989. ACM.

[63] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 166–173. ACM, 1989.

[64] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, Apr 1991.

[65] D. Duggan. Type-based hot swapping of running modules. *Acta Inf.*, 41(4):181–220, March 2005.

[66] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(2):146–170, 1995.

[67] J. M. Purtilo and J. M. Atlee. Module reuse by interface adaptation. *Software: Practice and Experience*, 21(6):539–556, 1991.

[68] B. Morel and P. Alexander. Spartacas: automating component reuse and adaptation. *IEEE Transactions on Software Engineering*, 30(9):587–600, 2004.

[69] J. Penix and P. Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535–542, 1997.

[70] J. Penix, P. Baraona, and P. Alexander. Classification and retrieval of reusable components using semantic features. In *Knowledge-Based Software Engineering Conference, 1995. Proceedings., 10th*, pages 131–138. IEEE, 1995.

[71] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997.

[72] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[73] R. Keller and U. Hölzle. Binary component adaptation. In *European Conference on Object-Oriented Programming*, pages 307–329. Springer, 1998.

[74] M. Nita and D. Grossman. Automatic Transformation of Bit-Level C Code to Support Multiple Equivalent Data Layouts. In L. Hendren, editor, *Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, pages 85–99, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[75] A. Podgurski and L. Pierce. Behavior sampling: a technique for automated retrieval of reusable components. In *Proceedings of the 14th international conference on Software engineering*, pages 349–361. ACM, 1992.

[76] H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[77] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 304–314, New York, NY, USA, 2002. ACM.

[78] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan. The superdiversifier: Peephole individualization for software protection. In K. Matsuura and E. Fujisaki, editors, *Advances in Information and Computer Security: Third International Workshop on Security, IWSEC 2008, Kagawa, Japan, November 25-27, 2008. Proceedings*, pages 100–120, Berlin, Heidelberg, 2008. Springer.

[79] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980.

[80] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *International Computer Software and Applications Conference*, 1977.

[81] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, pages 3–9, 1978.

[82] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, January 1986.

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

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

[85] B. Baudry and M. Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Comput. Surv.*, 48(1):16:1–16:26, September 2015.

[86] I. Gashi, P. Popov, and L. Strigini. Fault diversity among off-the-shelf sql database servers. In *Dependable Systems and Networks, 2004 International Conference on*, pages 389–398, June 2004.

[87] J. Han, D. Gao, and R. H. Deng. On the effectiveness of software diversity: A systematic study on real-world vulnerabilities. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 127–146, Berlin, Heidelberg, 2009. Springer-Verlag.

[88] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[89] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz. Runtime defense against code injection attacks using replicated execution. *Dependable and Secure Computing, IEEE Transactions on*, 8(4):588–601, July 2011.

[90] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[91] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[92] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[93] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[94] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.

[95] W. Sun, L. Xu, and S. Elbaum. Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics.

In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 79–89, New York, NY, USA, 2017. ACM.

[96] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–446, 2008.

[97] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)s*, pages 209–224, 2008.

[98] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. JDart: A dynamic symbolic analysis framework. In M. Chechik and J.-F. Raskin, editors, *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 442–459. Springer, 2016.

[99] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, Sep 2013.

[100] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, 2012.

[101] D. Babic, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*, pages 12–22, 2011.

[102] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, May 2012.

[103] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[104] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, pages 76–92, 2009.

[105] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[106] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[107] V. Sharma, M. W. Whalen, S. McCamant, and W. Visser. Veritesting challenges in symbolic execution of java. *SIGSOFT Softw. Eng. Notes*, 42(4):1–5, January 2018.

[108] V. Sharma, M. W. Whalen, S. McCamant, and W. Visser. Veritesting challenges in symbolic execution of Java. *SIGSOFT Softw. Eng. Notes*, 42(4):1–5, January 2018.

[109] K. Sen, G. Necula, L. Gong, and W. Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853, New York, NY, USA, 2015. ACM.

[110] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr 2003.

[111] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík. JBMC: a bounded model checking tool for verifying Java bytecode. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 183–190, 2018.

[112] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[113] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 47–54, 2007.

[114] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[115] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM.

[116] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *The 2015 Network and Distributed System Security Symposium*, 02 2015.

[117] WALA. `http://wala.sourceforge.net/wiki/index.php/Main_Page`. Accessed: 2018-11-16.

[118] S. Hussein, V. Sharma, M. W. Whalen, S. McCamant, and W. Visser. Formal Semantics of Java Ranger. `https://www-users.cs.umn.edu/~husse200/Semantics1_11_2019.pdf`, 2019–2020.

[119] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[120] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

[121] Java PathFinder team. Model Java Interface. `https://github.com/javapathfinder/jpf-core/wiki/Model-Java-Interface`. Accessed: 2019-01-18.

[122] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[123] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang. Dependence guided symbolic execution. *IEEE Transactions on Software Engineering*, 43(3):252–271, March 2017.

[124] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 144–154, New York, NY, USA, 2012. ACM.

[125] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. In *Computer Aided Verification (CAV)*, volume 10981 of *LNCS*, pages 183–190. Springer, 2018.

[126] D. Beyer. Automatic verification of c and java programs: Sv-comp 2019. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–155, Cham, 2019. Springer International Publishing.

[127] Herb Krasner. The Cost Of Poor Quality Software In The US: a 2018 report. `https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/index.htm`.

[128] WHOSE CURVE IS IT ANYWAY. https://whosecurve.com/.

[129] CVE-2020-0601 — Windows CryptoAPI Spoofing Vulnerability. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2020-0601.

[130] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium*, pages 745–761, August 2018.

[131] The Cost Of Fixing An Application Vulnerability. https://www.darkreading.com/risk/the-cost-of-fixing-an-application-vulnerability/d/d-id/1131049.

[132] How much do bugs cost to fix during each phase of the SDLC? https://www.synopsys.com/blogs/software-security/cost-to-fix-bugs-during-each-sdlc-phase/.

[133] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[134] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[135] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[136] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[137] G. Novark and E. D. Berger. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[138] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson. Zipr: Efficient static binary rewriting for security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566, June 2017.

[139] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, N. Stephens, R. Wang, and G. Vigna. Mechanical phish: Resilient autonomous hacking. *IEEE Security & Privacy*, 16(02):12–22, mar 2018.

[140] B. D. Rodes, A. Nguyen-Tuong, J. D. Hiser, J. C. Knight, M. Co, and J. W. Davidson. Defense against stack-based attacks using speculative stack layout transformation. In S. Qadeer and S. Tasiran, editors, *Runtime Verification*, pages 308–313, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[141] W. H. Hawkins, J. D. Hiser, and J. W. Davidson. Dynamic canary randomization for improved software security. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, CISRC '16, pages 9:1–9:7, New York, NY, USA, 2016. ACM.

[142] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[143] C. Le Goues, A. Nguyen-Tuong, H. Chen, J. W. Davidson, S. Forrest, J. D. Hiser, J. C. Knight, and M. Van Gundy. Moving target defenses in the helix self-regenerative architecture. In *Moving target defense II*, pages 117–149. Springer, 2013.

[144] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54, 2012.

[145] M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. L. Frazier, T. M. Frazier, B. Dutertre, I. Mason, N. Shankar, and

S. Forrest. Double helix and raven: A system for cyber fault tolerance and recovery. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, CISRC '16, pages 17:1–17:4, New York, NY, USA, 2016. ACM.

[146] Cyber Fault-tolerant Attack Recovery (CFAR). `https://www.darpa.mil/program/cyber-fault-tolerant-attack-recovery`, 2015–2018.

[147] S. Wang, P. Wang, and D. Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 236–247, March 2016.

[148] S. Wang, P. Wang, and D. Wu. Reassembleable Disassembling. In *24th USENIX Security Symposium*, pages 627–642, 2015.

[149] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[150] E. Bauman, Z. Lin, K. W. Hamlen, A. M. Mustafa, G. Ayoade, K. Al-Naami, L. Khan, K. W. Hamlen, B. M. Thuraisingham, F. Araujo, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 25th Network and Distributed Systems Security Symposium (NDSS)*, volume 12, pages 40–47. Springer, 2018.

[151] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[152] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not.*, 50(6):43–54, June 2015.

[153] T. Ji, L. Chen, X. Mao, and X. Yi. Automated program repair by using similar code containing fix ingredients. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 197–202, June 2016.

[154] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. Does genetic programming work well on automated program repair? In *2013 International Conference on Computational and Information Sciences*, pages 1875–1878, June 2013.

[155] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[156] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press.

[157] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[158] X. Meng and B. P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 24–35, New York, NY, USA, 2016. Association for Computing Machinery.

[159] A. Afzal, M. Motwani, K. Stolee, Y. Brun, and C. Le Goues. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[160] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[161] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 180–182, Piscataway, NJ, USA, 2017. IEEE Press.

[162] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. How developers debug software the dbgbench dataset: Poster. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 244–246, Piscataway, NJ, USA, 2017. IEEE Press.

[163] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, Dec 2015.

[164] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun. REPT: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 17–32, Carlsbad, CA, October 2018. USENIX Association.

[165] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao. Pomp: Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 17–32, USA, 2017. USENIX Association.

[166] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 529–540, New York, NY, USA, 2016. Association for Computing Machinery.

[167] W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1787–1804, Santa Clara, CA, August 2019. USENIX Association.

[168] Defense Advanced Research Projects Agency – Cyber Grand Challenge. `https://www.darpa.mil/program/cyber-grand-challenge`.

[169] Trail of Bits. DARPA Challenges Sets for Linux, Windows, and macOS . `https://github.com/trailofbits/cb-multios`.

[170] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery.

[171] N. Emamdoost, V. Sharma, T. Byun, and S. McCamant. Binary mutation analysis of tests using reassembleable disassembly. In *Binary Analysis Research Workshop*, Feb 2019.

Appendix A

Adapter Synthesis

A.0.1 Algorithm for Adapter Synthesis

Algorithm 2: Counterexample-guided adapter synthesis

Input : Target T as a code fragment or a function, reference function R, and adapter family $\mathcal{F}_{\mathcal{A}}$

Output: (input adapter $\mathcal{A}_{in}$, output adapter $\mathcal{A}_{out}$) or *null*

```
[1]  A_in ← default-input-adapter;
[2]  A_out ← default-output-adapter;
[3]  test-list ←empty-list;
[4]  while true do
[5]      counterexample ← CheckAdapter (A_in, A_out);
[6]      if counterexample is null then
[7]          return (A_in, A_out);
[8]      else
[9]          test-list.append(counterexample);
[10]     end
[11]     (A_in, A_out) ← SynthesizeAdapter (test-list);
[12]     if A_in is null then
[13]         return null;
[14]     end
[15] end
```

Here we present the pseudocode for our adapter synthesis algorithm, which was summarized by Figure 2.1 in Section 2.2.1. Algorithm 2 presents the main adapter synthesis

loop. This loop makes use of two procedures, `CheckAdapter` and `SynthesizeAdapter`, that are presented in Algorithms 3 and 4 respectively. We have implemented both the deductive procedures, `CheckAdapter` and `SynthesizeAdapter`, as calls to a symbolic executor.

Given an initial adapter called the *default* adapter for the target's inputs and outputs, Algorithm 2 first tries to find a counterexample for it on line 5. If no counterexample exists, then Algorithm 2 concludes that the default adapter must be correct on line 7. If a counter-example is found, Algorithm 2 adds it to a list of counterexamples on line 9. It then tries to synthesize an adapter for all previously found counterexamples by calling `SynthesizeAdapter` on line 11. If no adapter exists, then Algorithm 2 stops with this conclusion on line 13. Otherwise, it once again tries to find a counterexample for the latest adapter on line 5.

Algorithm 3 presents the `CheckAdapter` procedure that generates a counterexample for a given adapter. Given a concrete adapter for adapting the target's inputs and outputs, `CheckAdapter` runs the test harness presented in Listing 2.7 using unconstrained symbolic inputs. Since `CheckAdapter` uses a concrete adapter, if it can find any execution path through the test harness that causes the outputs of the target to not be equal to the adapted reference outputs, then it concretizes symbolic inputs on that execution path and reports the concretized input values as a counterexample.

Algorithm 4 presents the `SynthesizeAdapter` procedure that generates candidate adapters. Given a list of previously found counterexamples, `SynthesizeAdapter` runs the test harness described in Listing 2.7 with symbolic adapter formulas adapted the reference's inputs and outputs. If `SynthesizeAdapter` can find an execution path where all target outputs are equal to adapted reference outputs for all counterexamples in the input list, then it concretizes the adapter formulas into a concrete adapter and returns

it.

Algorithm 3: `CheckAdapter` procedure used by Algorithm 2. T and R are as defined in Algorithm 2.

Input : Concrete input adapter $\mathcal{A}_{in}$ and output adapter $\mathcal{A}_{out}$

Output: Counterexample to the given adapters or *null*

```
[1] args ← symbolic;
[2] while execution path available do
[3]     target-output ← execute T with input args;
[4]     reference-output ← execute R with input adapt(A_in, args);
[5]     if ! equivalent(target-output, adapt(A_out, reference-output)) then
[6]         return concretize(args);
[7]     end
[8] end
[9] return null;
```

Algorithm 4: `SynthesizeAdapter` procedure used by Algorithm 2. T and R are as defined in Algorithm 2. The form of the resulting adapters ($\mathcal{A}_{in}$, $\mathcal{A}_{out}$) is dictated by $\mathcal{F}_{\mathcal{A}}$.

Input : List of previously generated counterexamples test-list

Output: (input adapter $\mathcal{A}_{in}$, output adapters $\mathcal{A}_{out}$) or *null*

```
[1] A_in ← symbolic input adapter;
[2] A_out ← symbolic output adapter;
[3] while execution path available do
[4]     eq-counter ← 0;
[5]     while eq-counter < length(test-list) do
[6]         target-output ← execute T with input test;
[7]         reference-output ← execute R with input adapt(A_in, test));
[8]         if equivalent(target-output, adapt(A_out, reference-output)) then
[9]             eq-counter ← eq-counter + 1;
[10]        else
[11]            break;
[12]        end
[13]    end
[14]    if eq-counter == length(test-list) then
[15]        return (concretize(A_in), concretize(A_out));
[16]    end
[17] end
[18] return null;
```

A.0.2 Expanded results for reverse engineering evaluation

Table A.1: Metrics for adapters for the first 12 of the 24 reference functions using the type conversion adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
clamp(#3)	2171	8.9	51.7 (10.5)	7.1 (0.4)	1.4 (0.1)	44.6 (10.1)	18.4 (4.2)
prev_ pow_2(#1)	912	3.7	7.5 (0.6)	5.6 (0.4)	1.3 (0.1)	1.9 (0.2)	1.0 (0.1)
abs_diff(#2)	1511	6.4	14.9 (0.8)	9.1 (0.2)	1.8 (0.1)	5.8 (0.6)	2.0 (0.2)
bswap32(#1)	999	4.6	8.0 (0.3)	5.4 (0.1)	1.2 (0.0)	2.6 (0.2)	1.1 (0.1)
integer_ cmp(#2)	1025	4.8	10.4 (0.7)	6.6 (0.2)	1.4 (0.1)	3.8 (0.5)	1.6 (0.2)
even(#1)	892	3.9	7.2 (0.2)	5.0 (0.1)	1.2 (0.0)	2.2 (0.2)	1.1 (0.1)
div255(#1)	884	3.7	6.3 (0.2)	4.5 (0.1)	1.2 (0.0)	1.8 (0.1)	1.0 (0.1)
reverse_ bits(#1)	1179	4.8	11.1 (0.7)	7.1 (0.2)	1.5 (0.0)	4.0 (0.6)	1.7 (0.3)
binary_ log(#1)	963	3.8	8.0 (0.5)	5.0 (0.2)	1.2 (0.1)	3.0 (0.4)	2.3 (0.3)
median(#3)	1601	7.5	47.8 (15.2)	7.1 0.4)	1.5 (0.1)	40.7 (14.8)	18.1 (6.7)
hex_ value(#1)	880	3.6	6.5 (0.2)	4.8 (0.1)	1.0 (0.0)	1.7 (0.1)	1.3 (0.1)
get_descr_ len_ 24b(#1)	916	4.5	7.9 (0.3)	5.6 (0.1)	1.0 (0.0)	2.3 (0.1)	1.3 (0.1)

Table A.2: Metrics for adapters for the second set of 12 of the 24 reference functions using the type conversion adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
tile_ pos(#4)	6413	8.9	76.7 (32.3)	16.0 (3.6)	4.1 (2.0)	60.7 (28.7)	27.3 (15.3)
dirac_pic_ n_bef_ m(#2)	1275	6.9	15.8(1.6)	9.5 (0.3)	1.1 (0.1)	6.3 (1.2)	1.6 (0.2)
ps_id_ to_tk(#1)	894	3.8	4.4 (0.2)	2.6 (0.1)	1.0 (0.0)	1.8 (0.1)	1.3 (0.1)
leading_ zero_ count(#1)	946	4.8	5.6 (0.3)	3.1 (0.1)	1.0 (0.0)	2.5 (0.2)	1.3 (0.1)
trailing_ zero_ count(#1)	961	4.2	8.6 (0.6)	5.8 (0.2)	1.6 (0.2)	2.8 (0.3)	2.0 (0.2)
popcnt_32(#1)	894	4	7.1 (0.3)	5.0 (0.1)	1.3 (0.0)	2.1 (0.2)	1.0 (0.1)
parity(#1)	894	4	7.1 (0.2)	4.9 (0.1)	1.2 (0.0)	2.1 (0.1)	1.0 (0.1)
dv_audio_ 12_ to_16(#1)	894	3.4	6.9 (0.2)	5.3 (0.1)	1.4 (0.0)	1.7 (0.1)	1.0 (0.1)
is_power _2(#1)	894	4.1	8.4 (0.2)	6.2 (0.1)	1.5 (0.0)	2.2 (0.1)	1.1 (0.1)
Render RGB(#3)	1661	6.3	12.0 (0.6)	7.8 (0.2)	1.2 (0.0)	4.2 (0.4)	1.4 (0.2)
decode_ BCD(#1)	894	4.2	8.8 (0.4)	6.4 (0.1)	1.4 (0.0)	2.4 (0.3)	1.1 (0.1)
mpga_get_ frame_ samples(#1)	902	3.6	8.2 (0.2)	6.0 (0.1)	1.6 (0.1)	2.2 (0.1)	1.2 (0.1)

Table A.3: Metrics for the insubstitutable conclusion for the first 12 of the set of 24 reference functions using the type conversion adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
clamp(#3)	43496	8	59.2 (11.3)	6.6 (0.7)	1.6 (0.1)	52.6 (10.6)	35.8 (7.2)
prev_ pow_2(#1)	45889	5.2	11.7 (1.3)	7.2 (0.5)	2.4 (0.2)	4.6 (0.8)	2.3 (0.6)
abs_diff(#2)	45284	7	15.7 (1.1)	7.9 (0.3)	2.0 (0.1)	7.8 (0.9)	3.3 (0.4)
bswap32(#1)	45801	5.3	10.5 (0.5)	6.4 (0.2)	2.1 (0.1)	4.2 (0.3)	2.0 (0.2)
integer_ cmp(#2)	45628	5.8	26.9 (3.9)	5.6 (0.4)	1.7 (0.1)	21.3 (3.5)	16.6 (2.9)
even(#1)	45910	5	12.3 (0.8)	6.1 (0.2)	2.1 (0.1)	6.2 (0.6)	4.0 (0.5)
div255(#1)	45917	5.1	10.5 (0.5)	6.4 (0.2)	2.2 (0.1)	4.2 (0.3)	2.0 (0.2)
reverse_ bits(#1)	45621	5.5	13.6 (1.1)	7.0 (0.3)	2.2 (0.1)	6.6 (0.9)	3.3 (0.5)
binary_ log(#1)	45654	4.6	41.8 (8.7)	4.9 (0.9)	1.6 (0.1)	36.8 (7.7)	34.3 (7.5)
median(#3)	41300	7	117.1 (37.1)	6.0 (0.5)	1.6 (0.1)	111.1 (36.6)	94.8 (31.2)
hex_ value(#1)	45632	4.6	16.0 (2.4)	5.9 (0.9)	1.8 (0.1)	10.0 (1.6)	8.2 (1.5)
get_descr_ len_ 24b(#1)	45596	5.4	12.2 (1.7)	7.9 (1.3)	1.9 (0.1)	4.3 (0.3)	2.0 (0.2)

Table A.4: Metrics for the insubstitutable conclusion for the second set of 12 of 24 reference functions using the type conversion adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
tile_ pos(#4)	24579	7.1	100.9 (39.3)	10.2 (0.6)	2.8 (0.1)	90.7 (38.7)	70.2 (31.0)
dirac_pic_ n_bef_ m(#2)	45473	7.1	16.7 (1.6)	8.2 (0.4)	1.9 (0.1)	8.5 (1.2)	3.5 (0.4)
ps_id_ to_tk(#1)	45621	5	12.0 (2.2)	3.8 (1.2)	1.1 (0.1)	8.2 (1.1)	6.2 (0.9)
leading_ zero_ count(#1)	45812	5.2	7.7 (1.1)	3.6 (0.8)	1.1 (0.1)	4.1 (0.3)	2.0 (0.2)
trailing_ zero_ count(#1)	45782	4.9	41.8 (9.3)	5.1 (0.8)	1.6 (0.1)	36.7 (8.5)	34.1 (8.3)
popcnt_ 32(#1)	45909	5.3	10.8 (0.6)	6.5 (0.2)	2.1 (0.1)	4.3 (0.4)	2.0 (0.2)
parity(#1)	45911	5.3	11.1 (0.5)	6.6 (0.2)	2.2 (0.1)	4.5 (0.3)	2.1 (0.2)
dv_audio_ 12_ to_16(#1)	45803	5	15.6 (2.1)	5.3 (0.4)	1.8 (0.1)	10.2 (1.8)	8.1 (1.6)
is_power _2(#1)	45886	4.9	12.7 (1.0)	5.2 (0.3)	1.9 (0.1)	7.5 (0.8)	5.5 (0.7)
Render RGB(#3)	45138	5.3	10.3 (0.8)	6.1 (0.4)	2.1 (0.2)	4.2 (0.4)	2.0 (0.2)
decode_ BCD(#1)	45900	5.3	10.5 (0.9)	5.9 (0.2)	1.9 (0.1)	4.6 (0.6)	2.2 (0.4)
mpga_get_ frame_ samples(#1)	45718	4.9	15.0 (2.0)	6.2 (1.0)	1.9 (0.1)	8.8 (1.0)	6.6 (0.9)

Table A.5: Metrics for the timeout conclusion for the first 12 of the 24 reference functions using the type conversion adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
clamp(#3)	1156	16.4	600.0 (193.2)	95.0 (72.4)	78.3 (67.5)	505.0 (120.7)	261.4 (60.1)
prev_ pow_2(#1)	22	2.6	600.0 (594.4)	599.0 (594.4)	597.0 (594.0)	1.0 (0.1)	1.0 (0.1)
abs_diff(#2)	28	1.2	600.0 (595.9)	599.9 (595.9)	599.8 (595.9)	0.1 (0.0)	0.1 (0.0)
bswap32(#1)	23	2.6	600.0 (594.4)	599.1 (594.3)	597.0 (594.1)	0.9 (0.1)	0.9 (0.1)
integer_ cmp(#2)	170	3.7	600.0 (508.6)	546.7 (497.2)	537.7 (492.0)	53.3 (11.4)	22.0 (4.9)
even(#1)	21	2.5	600.0 (566.2)	599.0 (566.1)	597.1 (565.8)	1.0 (0.1)	1.0 (0.1)
div255(#1)	22	2.5	600.0 (583.5)	599.0 (583.4)	596.4 (583.1)	1.0 (0.1)	1.0 (0.1)
reverse_ bits(#1)	23	2.6	600.0 (594.0)	598.7 (593.9)	596.5 (593.6)	1.3 (0.1)	1.3 (0.1)
binary_ log(#1)	206	1	600.0 (591.0)	600.0 (591.0)	600.0 (591.0)	0.0 (0.0)	0.0 (0.0)
median(#3)	3922	14.3	600.0 (245.8)	46.5 (31.4)	33.6 (29.4)	553.5 (214.4)	352.8 (133.4)
hex_value(#1)	311	1.1	600.0 (502.6)	599.9 (502.6)	594.2 (497.6)	0.1 (0.0)	0.1 (0.0)
get_descr_ len_24b(#1)	311	1	600.0 (575.7)	600.0 (575.7)	598.1 (573.9)	0.0 (0.0)	0.0 (0.0)

Table A.6: Metrics for the timeout conclusion for the second 12 of the 24 reference functions using the type conversion adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
tile_ pos(#4)	15831	6.9	600.0 (571.3)	14.0 (2.8)	6.6 (2.3)	586.0 (568.5)	523.4 (516.5)
dirac_pic_n_ bef_m(#2)	75	16	600.0 (484.2)	530.7 (454.7)	511.9 (453.6)	69.3 (29.4)	4.1 (1.0)
ps_id_ to_tk(#1)	308	1.2	600.0 (596.9)	599.6 (596.8)	576.7 (574.0)	0.4 (0.0)	0.3 (0.0)
leading_ zero_ count(#1)	65	1.3	600.0 (596.9)	599.7 (596.9)	563.2 (560.5)	0.3 (0.0)	0.2 (0.0)
trailing_ zero_ count(#1)	80	1.9	600.0 (563.7)	599.1 (563.6)	583.8 (548.9)	0.9 (0.1)	0.6 (0.1)
popcnt_ 32(#1)	20	2.4	600.0 (594.3)	599.2 (594.2)	597.0 (594.0)	0.8 (0.1)	0.8 (0.1)
parity(#1)	18	2.4	600.0 (573.6)	598.9 (573.5)	579.6 (556.4)	1.1 (0.1)	1.1 (0.1)
dv_audio_ 12_ to_16(#1)	126	1.3	600.0 (375.1)	599.8 (375.1)	599.5 (375.0)	0.2 (0.0)	0.2 (0.0)
is_power _2(#1)	43	2.3	600.0 (557.3)	599.1 (557.2)	592.9 (557.0)	0.9 (0.1)	0.9 (0.1)
Render RGB(#3)	24	2.2	600.0 (465.0)	599.2 (464.9)	598.2 (464.9)	0.8 (0.1)	0.8 (0.1)
decode_ BCD(#1)	29	2.2	600.0 (471.9)	599.2 (471.8)	597.9 (471.6)	0.8 (0.1)	0.8 (0.1)
mpga_get_ frame_ samples(#1)	203	1.1	600.0 (554.7)	599.9 (554.7)	596.7 (554.5)	0.1 (0.0)	0.1 (0.0)

Table A.7: Metrics for adapters for the first 12 of 24 reference functions using the arithmetic adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
clamp(#3)	5450	9.4	215.0 (162.5)	33.4 (0.0)	8.4 (0.0)	181.7 (162.5)	76.7 (70.0)
prev_ pow_2(#1)	719	4.4	44.8 (21.4)	20.4 (0.0)	7.5 (0.0)	24.4 (21.4)	8.2 (7.3)
abs_ diff(#2)	1012	3.5	22.2 (8.7)	11.6 (0.0)	5.2 (0.0)	10.5 (8.7)	5.1 (4.3)
bswap32(#1)	968	3.3	25.8 (9.7)	14.4 (0.0)	6.1 (0.0)	11.4 (9.7)	5.4 (4.7)
integer_ cmp(#2)	924	3.2	32.8 (15.5)	15.4 (0.0)	7.6 (0.0)	17.3 (15.5)	9.5 (8.7)
even(#1)	1205	4	39.0 (19.8)	16.4 (0.0)	6.7 (0.0)	22.7 (19.8)	11.1 (9.8)
div255(#1)	880	3.3	26.9 (10.1)	14.9 (0.0)	6.6 (0.0)	11.9 (10.1)	5.4 (4.6)
reverse_ bits(#1)	1190	4.1	31.9 (13.1)	16.3 (0.0)	5.8 (0.0)	15.6 (13.1)	7.0 (6.0)
binary_ log(#1)	905	3.5	34.6 (15.1)	17.5 (0.0)	5.7 (0.0)	17.1 (15.1)	10.7 (9.2)
median(#3)	3661	6.6	141.7 (111.8)	22.1 (0.0)	7.6 (0.0)	119.5 (111.8)	62.3 (59.1)
hex_ value(#1)	880	3.3	28.9 (11.1)	16.1 (0.0)	5.4 (0.0)	12.8 (11.1)	10.9 (9.4)
get_descr_ len_24b(#1)	906	3.8	27.1 (9.9)	15.1 (0.0)	4.0 (0.0)	11.9 (9.9)	8.6 (7.2)

Table A.8: Metrics for adapters for the second 12 of 24 reference functions using the arithmetic adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
tile_pos(#4)	4926	7.6	191.6 (158.0)	21.2 (0.0)	6.7 (0.0)	170.4 (158.0)	95.8 (89.9)
dirac_pic_ n_bef _m(#2)	1174	4.8	38.2 (16.7)	18.4 (0.0)	4.3 (0.0)	19.8 (16.7)	10.1 (8.5)
ps_id_ to_tk(#1)	917	4	36.1 (15.3)	18.6 (0.0)	5.8 (0.0)	17.5 (15.3)	12.6 (11.0)
leading_ zero_ count(#1)	924	4.3	33.2 (13.2)	17.3 (0.0)	4.2 (0.0)	15.9 (13.2)	9.5 (7.8)
trailing_ zero_ count(#1)	881	3.9	50.3 (25.0)	22.8 (0.0)	8.9 (0.0)	27.6 (25.0)	19.8 (18.0)
popcnt_32(#1)	916	3.5	29.3 (11.7)	15.4 (0.0)	6.5 (0.0)	13.8 (11.7)	5.9 (5.1)
parity(#1)	880	3.4	29.6 (11.9)	15.5 (0.0)	6.9 (0.0)	14.1 (11.9)	5.8 (5.0)
dv_audio_ 12_to _16(#1)	880	3	29.6 (14.2)	13.7 (0.0)	7.9 (0.0)	15.9 (14.2)	8.3 (7.5)
is_power_2(#1)	880	3	28.2 (11.2)	15.4 (0.0)	8.1 (0.0)	12.8 (11.2)	6.5 (5.8)
Render RGB(#3)	5852	5.3	34.5 (15.0)	16.3 (0.0)	4.2 (0.0)	18.2 (15.0)	9.3 (7.8)
decode_ BCD(#1)	920	3.8	33.8 (14.3)	16.8 (0.0)	6.6 (0.0)	17.0 (14.3)	6.9 (5.9)
mpga_get_ frame_ samples(#1)	878	3.1	31.1 (14.6)	14.7 (0.0)	8.5 (0.0)	16.4 (14.6)	8.1 (7.4)

Table A.9: Metrics for the insubstitutable conclusion for the first 12 of the 24 reference functions using the arithmetic adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
clamp(#3)	22890	6.3	291.6 (240.8)	18.8 (0.0)	6.4 (0.0)	272.8 (240.8)	197.4 (174.4)
prev_ pow_2(#1)	35793	5.9	57.5 (33.3)	18.5 (0.0)	5.6 (0.0)	39.0 (33.3)	19.2 (16.8)
abs_ diff(#2)	45789	5.7	37.3 (22.2)	10.2 (0.0)	3.3 (0.0)	27.0 (22.2)	13.9 (11.8)
bswap32(#1)	45837	5.7	46.8 (26.3)	15.6 (0.0)	4.8 (0.0)	31.2 (26.3)	15.7 (13.6)
integer_ cmp(#2)	42234	4.9	147.1 (120.9)	12.7 (0.0)	4.9 (0.0)	134.4 (120.9)	115.7 (104.8)
even(#1)	45595	5.1	75.1 (54.1)	13.9 (0.0)	4.9 (0.0)	61.1 (54.1)	44.1 (39.6)
div255(#1)	45926	5.5	47.3 (26.5)	15.9 (0.0)	5.0 (0.0)	31.4 (26.5)	16.1 (13.9)
reverse_ bits(#1)	45614	5.7	49.7 (28.0)	16.2 (0.0)	4.9 (0.0)	33.5 (28.0)	16.8 (14.3)
binary_ log(#1)	7724	4	509.4 (467.1)	6.8 (0.0)	3.0 (0.0)	502.6 (467.1)	496.3 (462.1)
median(#3)	11300	4.3	335.2 (299.4)	9.6 (0.0)	4.1 (0.0)	325.6 (299.4)	307.8 (283.7)
hex_ value(#1)	39564	4.3	238.4 (209.4)	11.0 (0.0)	4.3 (0.0)	227.4 (209.4)	213.1 (197.0)
get_descr_ len_24b(#1)	45544	5.1	45.7 (24.1)	17.2 (0.0)	4.9 (0.0)	28.5 (24.1)	15.3 (13.3)

Table A.10: Metrics for the insubstitutable conclusion for the second 12 of the 24 reference functions using the arithmetic adapter. All times are reported in seconds.

function name	#	steps	total time (solver)	CE total time (solver)	CE last time (solver)	AS total time (solver)	AS last time (solver)
tile_pos(#4)	9971	6	292.6 (255.1)	14.0 (0.0)	4.6 (0.0)	278.6 (255.1)	241.5 (221.9)
dirac_pic_ n_bef _m(#2)	45551	5.9	52.6 (30.8)	16.4 (0.0)	4.8 (0.0)	36.2 (30.8)	17.2 (14.9)
ps_id_ to_tk(#1)	45289	5.2	111.9 (88.4)	14.3 (0.0)	5.2 (0.0)	97.6 (88.4)	78.8 (72.2)
leading_ zero_ count(#1)	45517	6	56.0 (32.4)	17.5 (0.0)	4.8 (0.0)	38.4 (32.4)	17.7 (15.1)
trailing_ zero_ count(#1)	6220	4.1	564.9 (517.1)	7.2 (0.0)	3.4 (0.0)	557.7 (517.1)	543.6 (504.6)
popcnt_32(#1)	45890	6	50.7 (29.2)	16.1 (0.0)	4.7 (0.0)	34.6 (29.2)	16.8 (14.5)
parity(#1)	45923	5.6	47.6 (27.1)	15.4 (0.0)	4.7 (0.0)	32.2 (27.1)	16.1 (13.9)
dv_audio_ 12_to_16(#1)	43426	5.2	200.5 (172.8)	12.5 (0.0)	4.8 (0.0)	188.0 (172.8)	155.2 (143.2)
is_power _2(#1)	45466	5	115.1 (91.9)	12.9 (0.0)	4.9 (0.0)	102.2 (91.9)	81.6 (73.9)
Render RGB(#3)	40948	5.3	43.9 (24.1)	15.3 (0.0)	5.0 (0.0)	28.6 (24.1)	15.3 (13.3)
decode_ BCD(#1)	45886	5.6	48.2 (27.3)	15.7 (0.0)	4.9 (0.0)	32.6 (27.3)	16.8 (14.5)
mpga_get_ frame_ samples(#1)	44453	5	154.3 (129.2)	12.1 (0.0)	4.5 (0.0)	142.1 (129.2)	122.1 (111.6)

Table A.11: Metrics for the timeout conclusion for the first 12 of 24 reference functions using the arithmetic substitution adapter. All times are reported in seconds.

function name	#	steps	total time	CE total time	CE last time	AS total time (solver)	AS last time (solver)
clamp(#3)	18483	10.8	600.0	44.2	16.3	555.8 (499.3)	299.5 (269.9)
prev_ pow_2(#1)	18	5.2	600.0	499.4	464.9	100.6 (96.2)	32.6 (30.1)
abs_diff(#2)	22	6.6	600.0	505.0	468.2	95.0 (84.0)	12.5 (10.2)
bswap32(#1)	18	4.6	600.0	564.6	557.1	35.4 (30.1)	10.3 (7.7)
integer_ cmp(#2)	3665	9.1	600.0	47.6	24.2	552.4 (514.3)	456.5 (427.8)
even(#1)	23	5.3	600.0	461.3	438.0	138.7 (121.1)	92.0 (79.4)
div255(#1)	17	2.6	600.0	594.1	587.7	5.9 (4.7)	5.9 (4.7)
reverse_ bits(#1)	19	3.9	600.0	570.7	539.1	29.3 (25.3)	9.3 (7.1)
binary_ log(#1)	38194	5	600.0	19.7	9.6	580.3 (544.3)	531.6 (499.4)
median(#3)	31862	8.5	600.0	30.0	12.1	570.0 (537.6)	392.9 (371.6)
hex_ value(#1)	6379	6.7	600.0	44.2	24.5	555.8 (519.4)	500.5 (468.8)
get_descr_ len_24b(#1)	373	1	600.0	600.0	600.0	0.0 (0.0)	0.0 (0.0)

Table A.12: Metrics for the timeout conclusion for the second 12 of 24 reference functions using the arithmetic substitution adapter. All times are reported in seconds.

function name	#	steps	total time	CE total time	CE last time	AS total time (solver)	AS last time (solver)
tile_pos(#4)	31926	8.8	600.0	23.2	6.4	576.8 (549.3)	408.8 (393.6)
dirac_pic_ n_bef _m(#2)	98	15.3	600.0	355.9	309.4	244.1 (216.7)	35.8 (29.9)
ps_id_ to_tk(#1)	617	3	600.0	506.5	495.5	93.5 (121.6)	73.6 (103.4)
leading_ zero_ count(#1)	382	17.5	600.0	331.4	235.0	268.6 (240.3)	25.2 (21.7)
trailing_ zero_ count(#1)	39698	5.1	600.0	19.0	7.8	581.0 (547.0)	533.1 (502.6)
popcnt_32(#1)	17	2.6	600.0	595.2	590.6	4.8 (3.7)	4.8 (3.7)
parity(#1)	20	6	600.0	526.0	501.1	74.0 (65.6)	17.7 (14.3)
dv_audio_ 12_ to_16(#1)	2517	9.3	600.0	41.5	16.6	558.5 (522.4)	436.5 (410.4)
is_power _2(#1)	477	11.8	600.0	87.4	42.5	512.6 (514.7)	266.7 (288.9)
Render RGB(#3)	23	3.2	600.0	591.6	564.7	8.4 (6.6)	6.3 (5.0)
decode_ BCD(#1)	17	2.6	600.0	594.4	588.8	5.6 (4.4)	5.6 (4.4)
mpga_get_ frame_ samples(#1)	1348	9.6	600.0	88.9	59.1	511.1 (488.5)	408.3 (394.6)

For the results reported in Section 2.4.7, we report detailed metrics for the three possible conclusions, adapter found, not substitutable, timed out, in the Tables A.1, A.2, A.3, A.4, A.5, A.6 respectively using the type conversion with return value substitution adapter. Similar results using the arithmetic with return value substitution adapter are reported in Tables A.7 and A.8, A.9 and A.10, A.11 and A.12. In the first column, after each reference function's name, the `#N` within parenthesis reports the number of arguments taken by the reference function. The "total time (solver)" column in Tables A.5, A.6 has a value of 600 followed by the amount of time spent in the solver. This first value is always 600 because every type conversion adapter synthesis was allowed to run with a 5 minute timeout as mentioned in Section 2.4.7. Similarly, the "total time (solver)" column in Tables A.11 and A.12 has a value of 600 followed by the amount of time spent in the solver because the arithmetic adapter evaluation was allowed a time budget of 600 seconds (10 minutes) per adapter synthesis task. Tables A.1, A.2 and A.7, show that most adapters are found in much less time than 600 seconds with a significant portion of the total time being taken by the last adapter search and counterexample search steps (in the "CE last time" and "AS last time" columns). Tables A.3, A.4 and A.9 and show that quite often the last adapter search step consumes a significant portion of the total running time. This observation can be explained by the fact that a failure of the last adapter search step is what leads adapter synthesis to conclude that no adapter exists in a given family of adapters. Tables A.5, A.6 and A.11, A.12 show that for the three functions with the most number of timeouts (`clamp`, `median`, and `tile_pos`), the adapter synthesis stopped during an adapter search step. This observation leads to the hypothesis that most timeouts would have led to a "not substitutable" conclusion as indicated by Figures 2.11, A.1, A.2. Tables A.5, A.6 and A.11, A.12 also show that for a number of reference functions, a significant portion of the total time budget was spent in solving.

A.0.3 Timeouts with `tile_pos` and `median`

Here we report the histograms of total running of adapter synthesis in tasks when an adapter was found. We report these histograms for the `tile_pos` and `median` reference functions when using the type conversion adapter with return value substitution. The similar histogram for the `clamp` reference function was reported in Figure 2.11 in Section 2.4.7. Figures A.1 and A.2 show the histogram of total running time for the "adapter found" conclusion for the `tile_pos` and `median` reference functions when using the type conversion adapter. We also report similar histograms for the `clamp`, `tile_pos`, `median`, and `trailing_zero_count` reference functions when using the arithmetic adapter. Figures A.3, A.4, A.2, and A.6 show the histogram of total running time for the "adapter found" conclusion for the `clamp`, `tile_pos`, `median`, and `trailing_zero_count` reference functions when using the arithmetic adapter with return value substitution. Figures A.1, A.2, A.3, A.4, A.2, and A.6 lend further support to the hypothesis that a 600 second timeout is sufficient for finding most adapters.

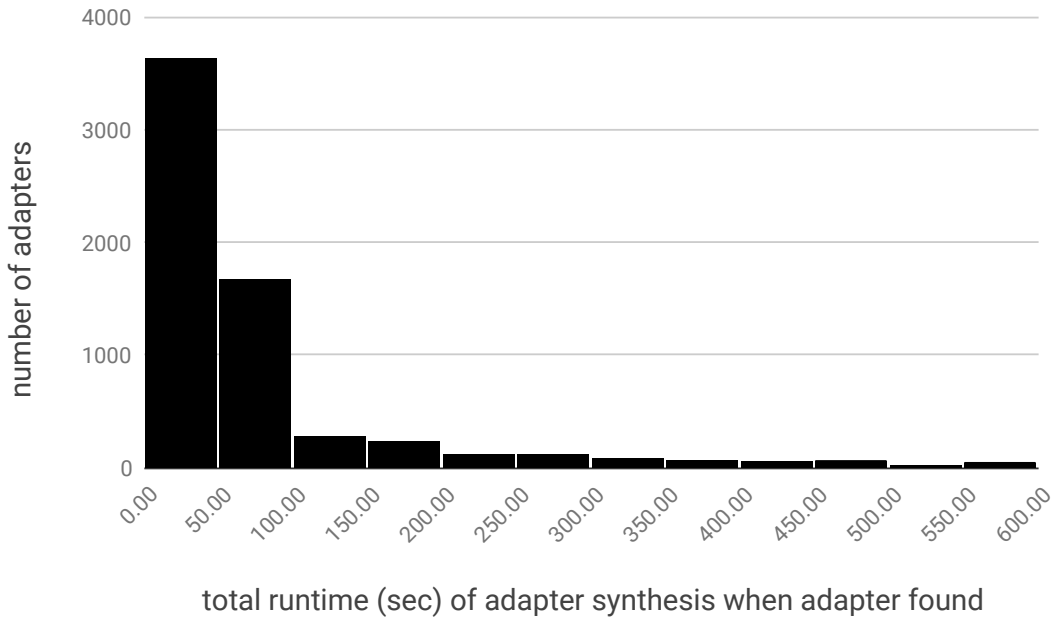


Figure A.1: Running times for synthesized adapters using `tile_pos` reference function when using the type conversion adapter

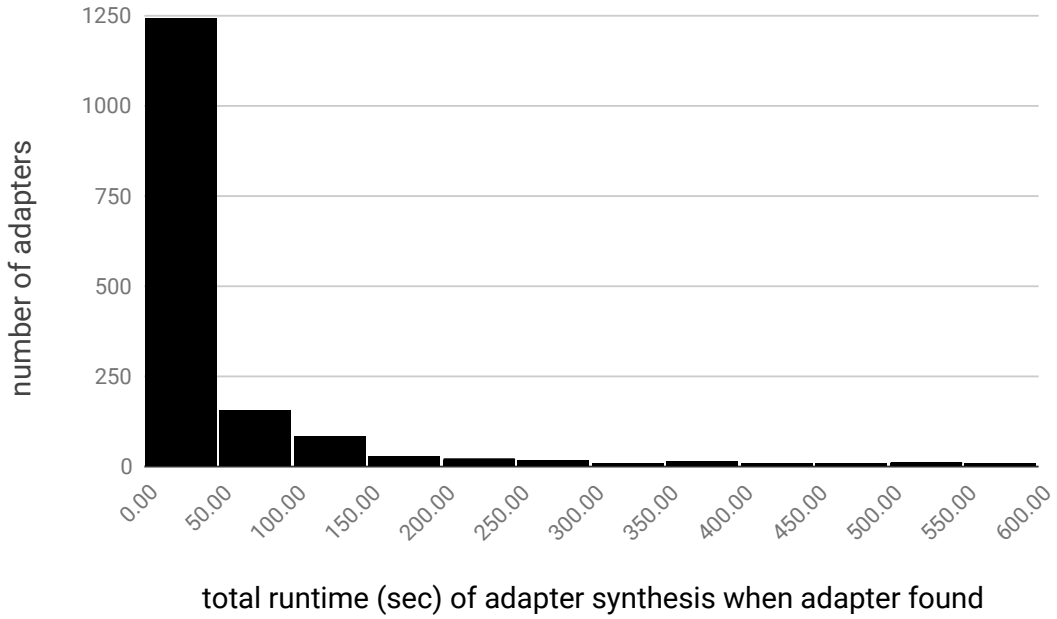


Figure A.2: Running times for synthesized adapters using `median` reference function when using the type conversion adapter

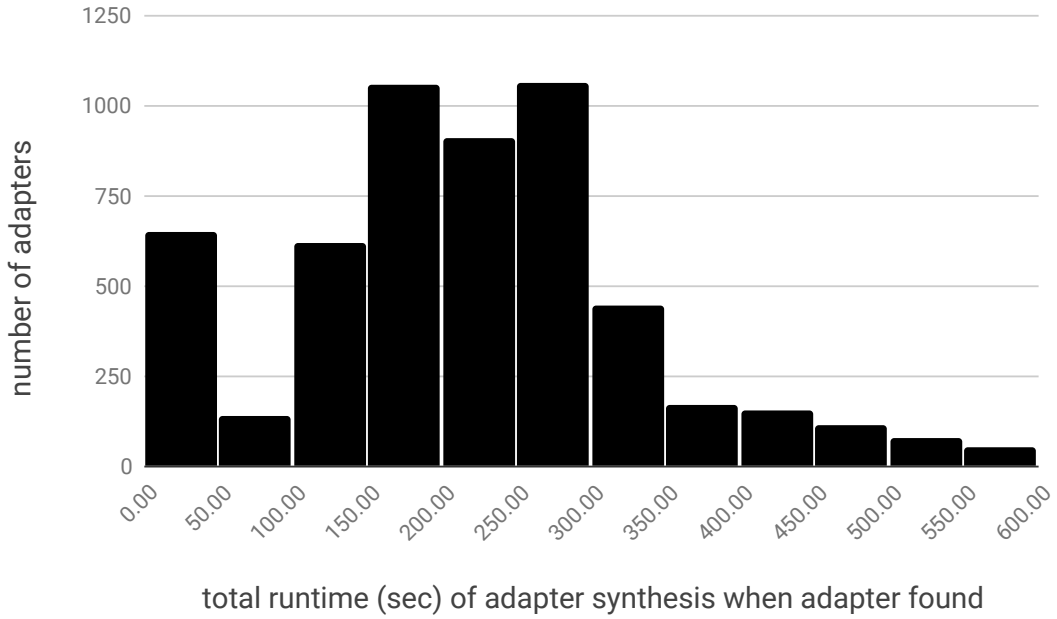


Figure A.3: Running times for synthesized adapters using `clamp` reference function when using the arithmetic adapter with return value substitution

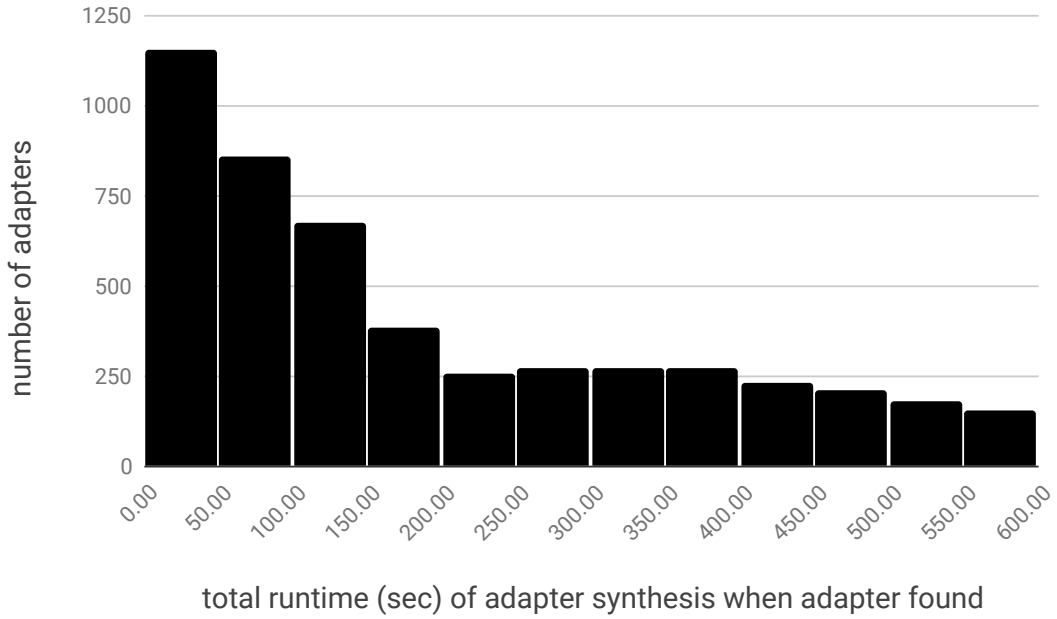


Figure A.4: Running times for synthesized adapters using `tile_pos` reference function when using the arithmetic adapter with return value substitution

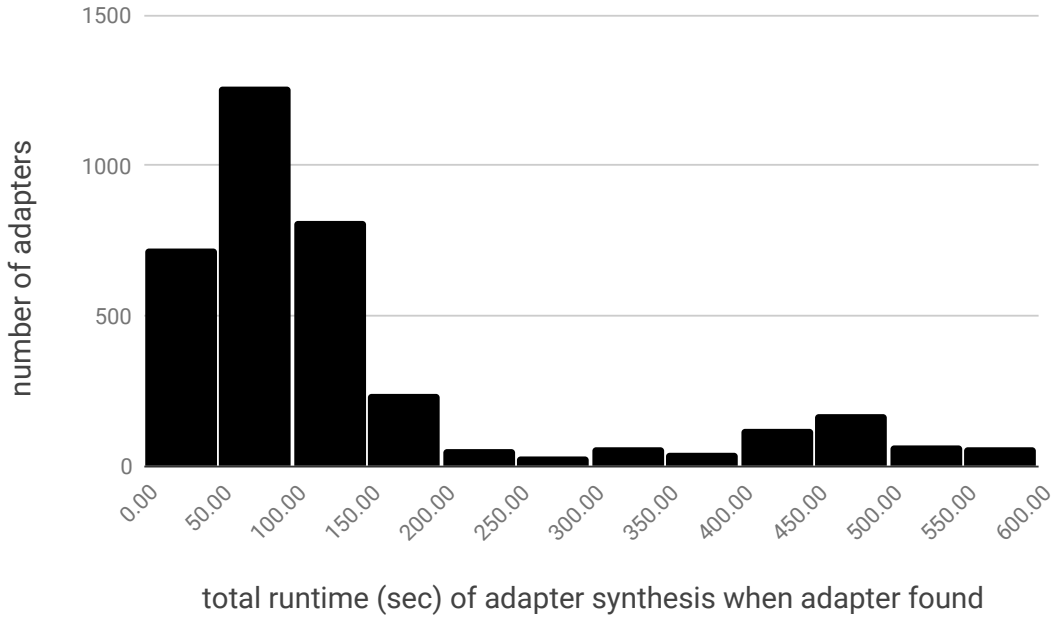


Figure A.5: Running times for synthesized adapters using `median` reference function when using the arithmetic adapter with return value substitution

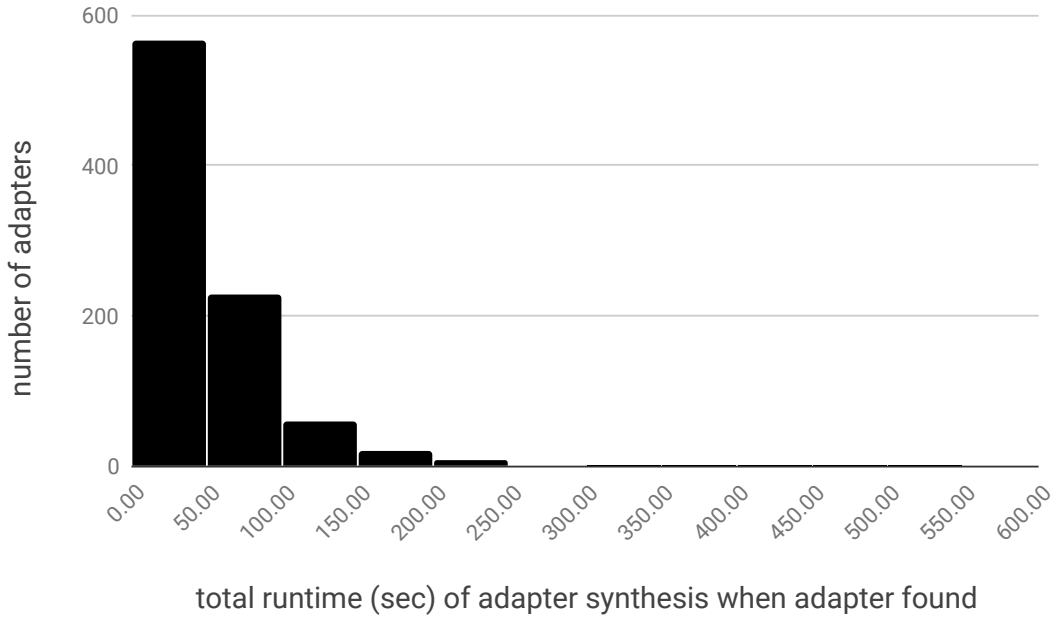


Figure A.6: Running times for synthesized adapters using `trailing_zero_count` reference function when using the arithmetic adapter with return value substitution

Appendix B

Program Repair

The following listing presents the `recv_delim` method that is called from Listing 5.1 in chapter 5. The `recv_delim` method reads up to `size` bytes from the standard input as long as it does not get the delimiter passed as an argument in `delim`.

```
1 int recv_delim(int fd, char *buf, const unsigned long size, char delim) {
2     unsigned long rx = 0, rx_now = 0;
3     while (rx < size) {
4         rx_now = read(fd, buf + rx, 1);
5         if (buf[rx] == delim || rx_now <= 0) break;
6         rx += rx_now;
7     }
8     return rx;
9 }
```

Listing B.1: A C method that receives input from a file descriptor `fd`