Abstract

Abstract

Contents

List of Tables

List of Figures

Chapter 1

Finding Substitutable Binary Code By Synthesizing Adapters

1.1 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 paper, we improve upon a previously presented technique [1] 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 [2].

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 1.1 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 signifies the input is a letter by returning 1024, while the musl implementation returns 1 in that case. 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 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.

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 $f_2 \leftarrow f_1$. To efficiently search for an adapter, we use counterexample guided inductive synthesis (CEGIS) [3]. 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.

```c
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 1.1: musl and glibc implementations of the *isalpha* predicate and a wrapper around the glibc implementation that makes it equivalent to the musl implementation

We implement adapter synthesis to adapt around inputs and outputs in registers and memory. 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 can be adaptably substituted by reference functions written for the VLC media player [4]. 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. 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 1.2 presents our algorithm for adapter synthesis and describes our adapter families. Section 1.3 describes our implementation, and Section 1.4 presents examples of application of adapter synthesis, large-scale evaluations, and a comparison of two adapter search implementations. Section 1.5 discusses limitations and future work, Section 1.6 describes related work, and Section 1.7 concludes.

1.2 Adapter Synthesis

1.2.1 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. Our adapter synthesis algorithm is summarized in Figure 1.1. As shown in Figure 1.1, 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 [3]. Although

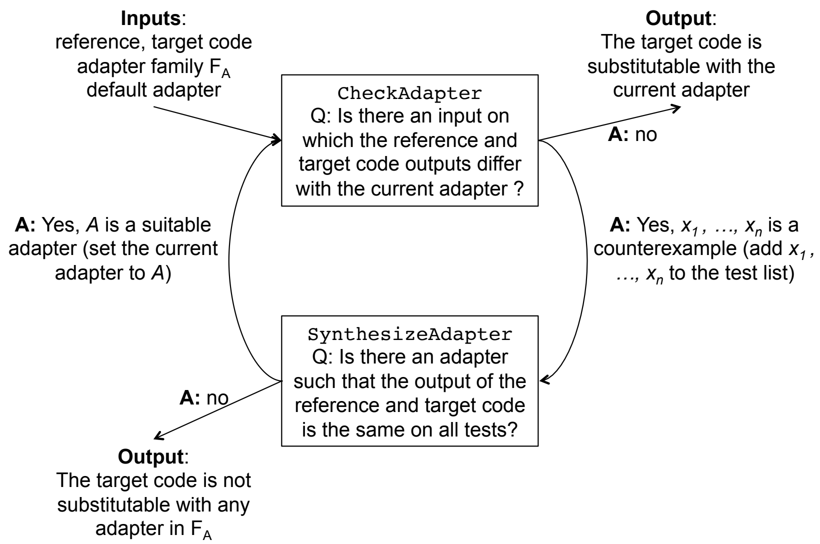


Figure 1.1: Counterexample-guided adapter synthesis

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 (see Section 1.4.5).

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. Algorithm 1 presents the main adapter synthesis loop, Algorithm 2 presents the `CheckAdapter` procedure that generates counterexamples, and Algorithm 3 presents the `SynthesizeAdapter` procedure that generates candidate adapters. `CheckAdapter` and `SynthesizeAdapter` are both implemented as calls to a symbolic executor.

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

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*

$\mathcal{A}_{in} \leftarrow$ default-input-adapter;
$\mathcal{A}_{out} \leftarrow$ default-output-adapter;
test-list $\leftarrow$empty-list;
while *true* **do**
    counterexample $\leftarrow$ `CheckAdapter` ($\mathcal{A}_{in}$, $\mathcal{A}_{out}$);
    if counterexample *is null* **then**
        return $(\mathcal{A}_{in}, \mathcal{A}_{out})$;
    else
        test-list.append(counterexample);
    end
    $(\mathcal{A}_{in}, \mathcal{A}_{out}) \leftarrow$ `SynthesizeAdapter` (test-list);
    if $\mathcal{A}_{in}$ *is null* **then**
        return *null;*
    end
end

Algorithm 1: Counterexample-guided adapter synthesis

timeouts in more detail in Section 1.4.5.

1.2.2 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 family of adapters allows replacement of any reference function argument by one of the target function arguments or a constant. 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 1.4.3.

Argument Substitution with Type Conversion

This 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

```
Input  : Concrete input adapter A_in and output adapter A_out
Output: Counterexample to the given adapters or null
args ← symbolic;
while execution path available do
    target-output ← execute T with input args;
    reference-output ← execute R with input adapt(A_in, args);
    if ! equivalent(target-output, adapt(A_out, reference-output)) then
        return concretize(args);
    end
end
return null;
```

Algorithm 2: `CheckAdapter` procedure used by Algorithm 1. T and R are as defined in Algorithm 1.

be produced by either a sign extension or zero extension on the low 8, 16, or 32 bits of the argument register. This adapter family also allows for converting target function arguments to boolean values by comparing those arguments to zero.

Arithmetic adapter

This family allows reference function arguments to be arithmetic combinations of target function arguments. This adapter family has been implemented by Hietala [5].

Memory Substitution

This family of adapters 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. 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 1.2. This makes our adapter synthesis more powerful because it can be used in combination with other adapter families that allow argument substitution. This adapter family synthesizes correct adapters between RC4 implementations in the mbedTLS and OpenSSL libraries in Section 1.4.2.

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 1.1. The wrapper function *adapted_isalpha* in Listing 1.1 performs return value substitution.

```
Input  : List of previously generated counterexamples test-list
Output: (input adapter A_in, output adapters A_out) or null
A_in ← symbolic input adapter;
A_out ← symbolic output adapter;
while execution path available do
    eq-counter ← 0;
    while eq-counter < length(test-list) do
        target-output ← execute T with input test;
        reference-output ← execute R with input adapt(A_in, test));
        if equivalent(target-output, adapt(A_out, reference-output)) then
            eq-counter ← eq-counter + 1;
        else
            break;
        end
    end
    if eq-counter == length(test-list) then
        return (concretize(A_in), concretize(A_out));
    end
end
return null;
```

Algorithm 3: `SynthesizeAdapter` procedure used by Algorithm 1. T and R are as defined in Algorithm 1. The form of the resulting adapters ($\mathcal{A}_{in}$, $\mathcal{A}_{out}$) is dictated by $\mathcal{F}_{\mathcal{A}}$.

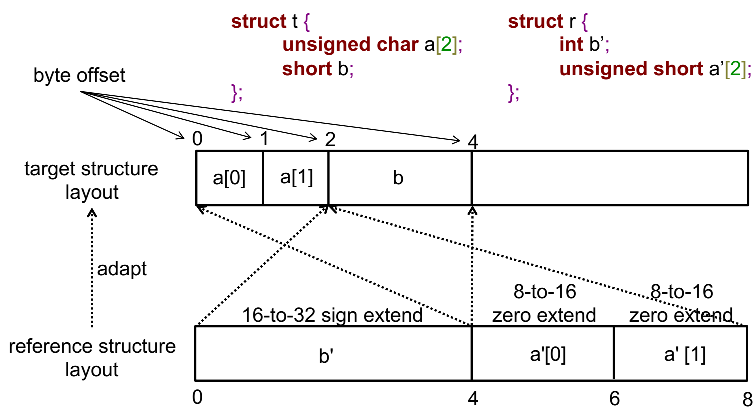
1.2.3 Example

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 1.2. 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 1.1), 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 1.1), we use symbolic execution to search for a new adapter $\mathcal{A}$ that will make `target(x)` equivalent

Figure 1.2: Memory substitution adapter to make *struct r* adaptably equivalent to *struct t*




```
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 target(y, 1, x, x);
}
```

Listing 1.2: Two implementations of similar adaptable functionality

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 $\mathcal{A}(x) = (y, 1, x, x)$.

1.2.4 Extensibility

The adapter synthesis algorithm presented in this section is not tied to a source programming language or adapter family. In our implementation (Section 1.3) 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 we work at the binary level, we are also not restricted to working at the level of target functions as described so far. In Section 1.4.5, 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.

1.3 Implementation

We implement adapter synthesis for Linux/x86-64 binaries using the symbolic execution tool FuzzBALL [6], which is freely available [7]. 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 [8] Vine library [9] and interfaces with the STP [10] 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.

1.3.1 Test Harness

To compare code for equivalence we use a test harness similar to the one used by Ramos et al. [11] 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 1.3 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 1.3: Test harness

We use the same test harness for both counterexample search (called `CheckAdapter` in Figure 1.1) and adapter search (called `SynthesizeAdapter` in Figure 1.1). 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 1.3.2) 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 1.3. For simplicity, Listing 1.3 shows only the return values r_1 and r_2 as being used for equivalence checking.

1.3.2 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 1.4. 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 1.4, 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.

```
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 1.4: Argument Substitution adapter

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 [12]. 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.

1.3.3 Equivalence checking of side-effects

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.

1.4 Evaluation

In this section, we evaluate the performance of adapter synthesis and present its applications using case studies. In Section 1.4.1, we present an example of finding adaptable substitutability modulo a bug—enabling programmers to more easily replace buggy components of their code. In Section 1.4.2, we show how our technique can enable programmers to switch between different libraries with the same functionality. In Section 1.4.3, we show that adapter synthesis can find adapters even within a library. In Section 1.4.4, we compare symbolic execution-based adapter search with concrete enumeration-based adapter search. Finally, in Section 1.4.5, we show how our technique can be used for reverse engineering.

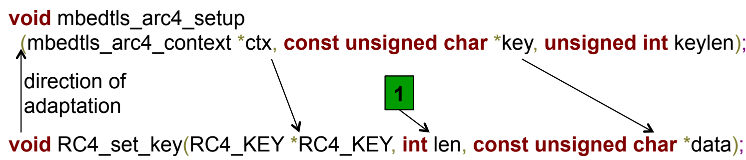
1.4.1 Case Study: Security

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

Consider a 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 1.5. 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 [13]. 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 1.5 to

Figure 1.3: Argument substitution adapter to make *RC4_set_key* adaptably equivalent to *mbedtls_arc4_setup*



not be satisfied, allowing execution to continue to line 5 and causing an out-of-bounds memory access. *lookup2* in Listing 1.5 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*. For this reason, 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*. Our adapter synthesis implementation synthesizes the correct argument substitution adapter in the *lookup1* ← *lookup2* direction in about 8 minutes. 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 1.5. This case study shows adapter synthesis can replace a buggy function with its bug-free variant by doing adaptation modulo a bug.

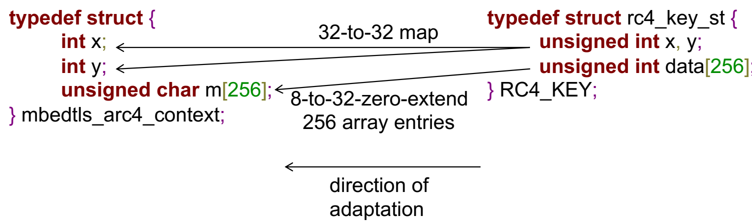
1.4.2 Case Study: Library Replacement

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 1.3. Executing each of these initialization routines requires 256 rounds of mixing bytes from the key string into the context. 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 1.4. 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)

Figure 1.4: Memory substitution adapter to make *RC4_KEY* adaptably equivalent to *mbedtls_arc4_context*



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 1.4. 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.

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 ($2 \times 4 + 256 \times 1$) byte mbedTLS context and 1032 ($2 \times 4 + 256 \times 4$) 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,

Table 1.1: 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

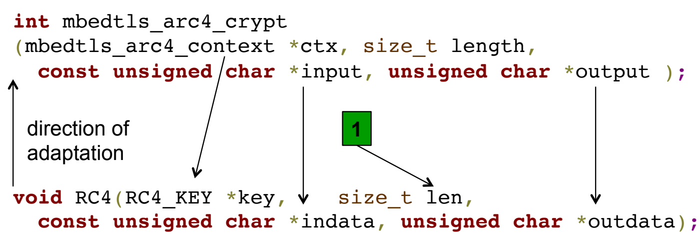
memory, and return value substitutions. It finds adaptable equivalence in both directions of adaptation by checking equivalence between side-effects on the structure objects (*ctx* for *mbedtls_arc4_setup*, *RC4_KEY* for *RC4_set_key*). The correct adapter for adaptably substituting the *mbedtls_arc4_setup* function with the *RC4_set_key* function is shown in Figure 1.3. 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 [14, 15] 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 1.1. The execution time shown in Table 1.1 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 1.1 and the observation that our adapter synthesis never used more than 1 CPU core and 4 GB of RAM, we estimated [16] the cost of this computation on a Amazon EC2 instance (t2-medium). Table 1.2 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 1.5.

Table 1.2: 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

Figure 1.5: Adapter performing argument and memory substitution to make *mbedtls_crypt* in the mbedTLS library adaptably substitutable by *RC4* in OpenSSL



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 1.5. 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 1.5. Our adapter synthesis tool finds the correct argument and memory substitution adapters in both directions of adaptation. Tables 1.1, 1.2 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 1.1 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.

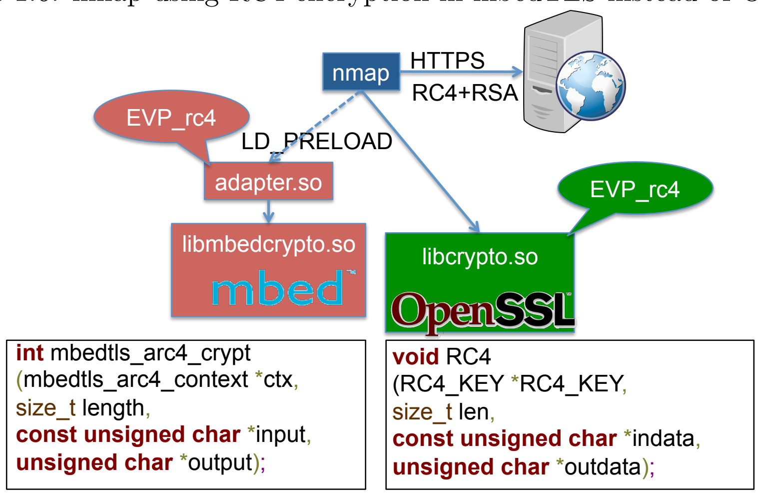
On improving memory substitution performance

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. [17]), 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.

RC4 adapter verification using nmap

We verified the correctness of our RC4 memory substitution adapter using nmap with the setup shown in Figure 1.6. 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

Figure 1.6: nmap using RC4 encryption in mbedTLS instead of OpenSSL



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 $\leftarrow$ mbedTLS structure adapter, was the same as the output of nmap when using the system OpenSSL library.

1.4.3 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

Table 1.3: Adapter Synthesis over 13130 function pairs without memory-based equivalence checking

adapter type	Inequiv.	adapters Found	Timeout	Target function crashed
arg. sub.	8887	382	3014	847
type conv.	8909	383	2989	849

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 and type conversion adapter families combined with the return value adapter family because these families scale well and are widely applicable. We ran our adapter synthesis with a 2 minute timeout on a machine running CentOS 6.8 with 64-bit Linux kernel version 2.6.32 using 64 GB RAM and a Intel Xeon E5-2680v3 processor. 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.

Results

Table 1.3 summarizes the results of searching for argument substitution and type conversion adapters with a return value adapter within the 13130 function pairs described above. 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.

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 30 interesting true positives shown in Table 1.4.

The remaining adapters found were correct, but trivial. The first column in Table 1.4 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.

Table 1.4: adapters found within eglibc-2.19

$f_1 \leftarrow f_2$ 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
ldiv(1) ↔ lldiv(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
putchar(1) ↔ putchar_unlocked putwchar(1) ↔ putwchar_unlocked(1)	#0
recv(4) ← recvfrom(6) send(4) ← sendto(6)	32-to-64S(#0), #1, #2, 32-to-64S(#3), 0, 0
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

The last three rows shown in Table 1.4 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.

1.4.4 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 1.2. We use the Pin [18] 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 1.4.3, 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 1.7. For concrete enumeration-based adapter search, Figure 1.7 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 1.7 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.

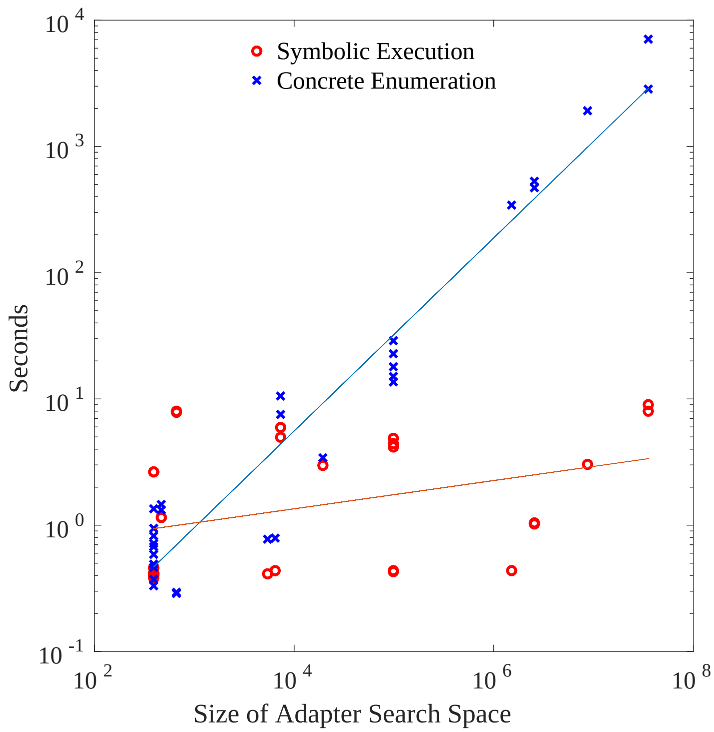
1.4.5 Large-Scale Reverse Engineering

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 a ARM firmware image and reference functions chosen from the source code of a popular media player.

Code fragment selection

Rockbox [19] 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

Figure 1.7: Comparing concrete enumeration-based adapter search with binary symbolic execution-based adapter search for adapters presented in Section 1.4.3



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. Intuitively, any code fragment that can be adaptably substituted by 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 our simple 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 [4] 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 $\times$ 24) adapter synthesis tasks, with each adapter synthesis search space consisting of 1.353×10^{127} adapters, too large for concrete enumeration. We set a 5 minute hard time limit and a total memory limit of 2 GB per adapter synthesis task. We split the adapter synthesis tasks for each reference function into 32 parallel jobs, creating a total of 768 (32 $\times$ 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. We present our results in Table 1.5. 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 full generality 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 full generality of the reference function for 11 reference functions. Reference functions for which we found no use of full generality are omitted in Table 1.5. 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 1.8 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

Table 1.5: Reverse engineering results using 46831 target code fragments from a Rockbox firmware image and 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.

	adapter				no adapter	timeout
fn_name	#(full)	#cluster	steps	total time	#	#
clamp	683 (177)	110	12.9	99.3	40553	5595
abs_diff	575 (5)	75	10.5	20.0	46250	6
bswap32	115 (8)	19	8.7	16.6	46708	8
integer_ cmp	93 (5)	15	9.6	21.4	46467	271
even	3 (2)	3	5.7	11.3	46823	5
median	332 (42)	60	13.7	119.2	32171	14328
get_ descr_len	22 (9)	2	9	16.7	46625	184
tile_pos	5617 (407)	909	10.9	53.5	24696	16518
dirac_pic_n _bef_m	330 (2)	18	13.2	25.7	46393	108
RenderRGB	763 (2)	64	10.8	27.5	46061	7
mpga_get_ frame_samples	22 (15)	4	5	7.9	46235	574

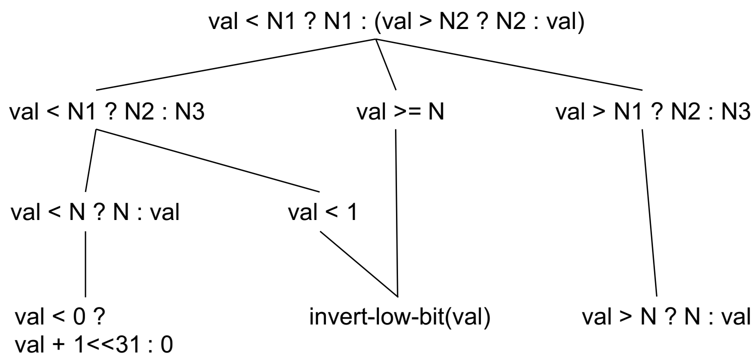


Figure 1.8: Subset of partial order relationship among adapted clamp instances

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.

Timeouts were the third possible conclusion of each adapter synthesis task. The number of timeouts is reported in Table 1.5. We show a histogram of the total running time used to find adapters in Figure 1.9 for the `clamp` reference function. Similar histograms for `tile_pos` and `median` reference functions can be found in Section 1.4.5.

Timeouts with `tile_pos` and `median`

Here we report the histograms of timeouts for the `tile_pos` and `median` reference functions. Please refer to Figures 1.9, 1.10 and 1.11. The number of adapters found after 300 seconds decreases rapidly, consistent with the mean total running time (subcolumn *total time* under column *adapter* in Table 1.5) of 99.3 seconds for the `clamp` reference function. Table 1.5 also shows that the total running time, when our tool concludes with finding an adapter, is significantly less than 300 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 300-second timeout was appropriate for this experiment, and that most timeouts would not have led to adapters.

1.4.6 Comparing adapter families

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

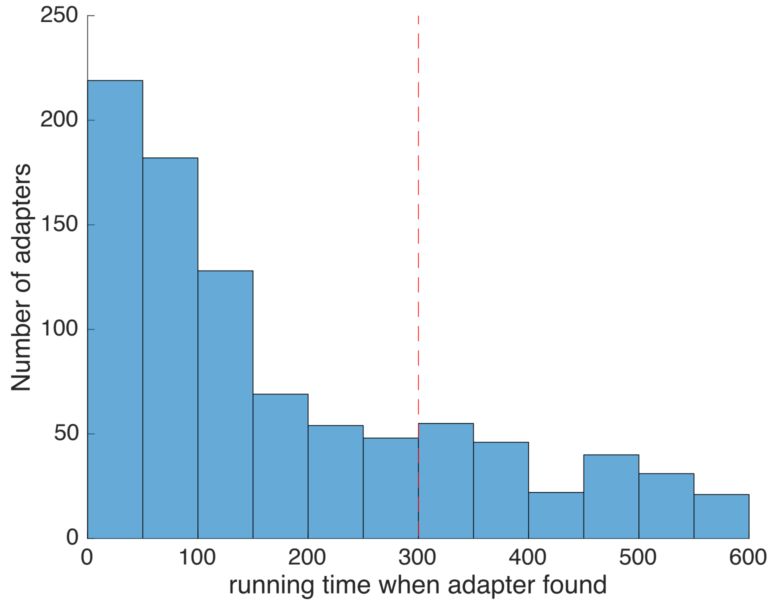


Figure 1.9: Running times for synthesized adapters using `clamp` reference function

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 1.6. As expected, the number of timeouts increases with an increase in the size of adapter search space. Table 1.6 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 1.6 suggests that, when computationally feasible, adapter families should be tried in increasing order of

Table 1.6: Comparing adapter families with 46,831 target code fragments and `clamp` reference function

	size	#-ad	#-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

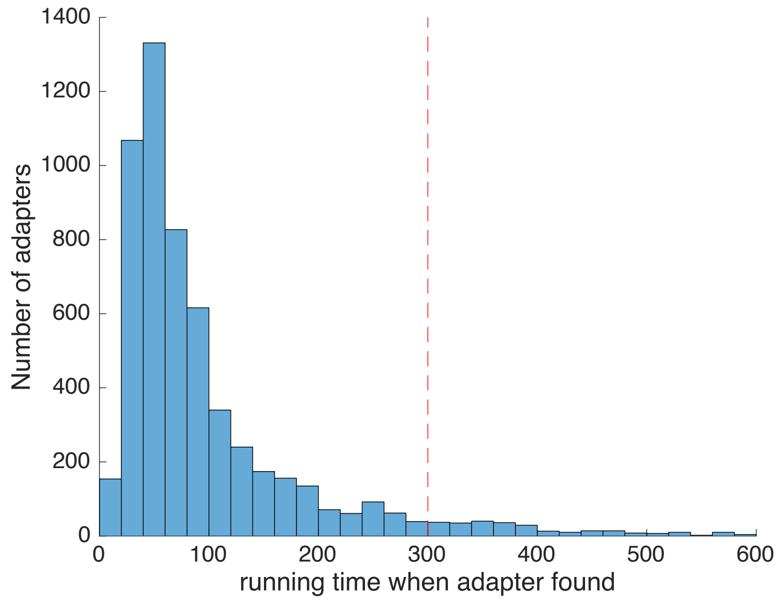


Figure 1.10: Running times for synthesized adapters using `tile_pos` reference function

expressiveness to have the fewest timeouts overall. We plan to explore this tradeoff between expressiveness and effectiveness of adapter families in the future.

1.5 Limitations and Future Work

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.

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.

Our tool currently assumes that all behaviors of the target function must be matched,

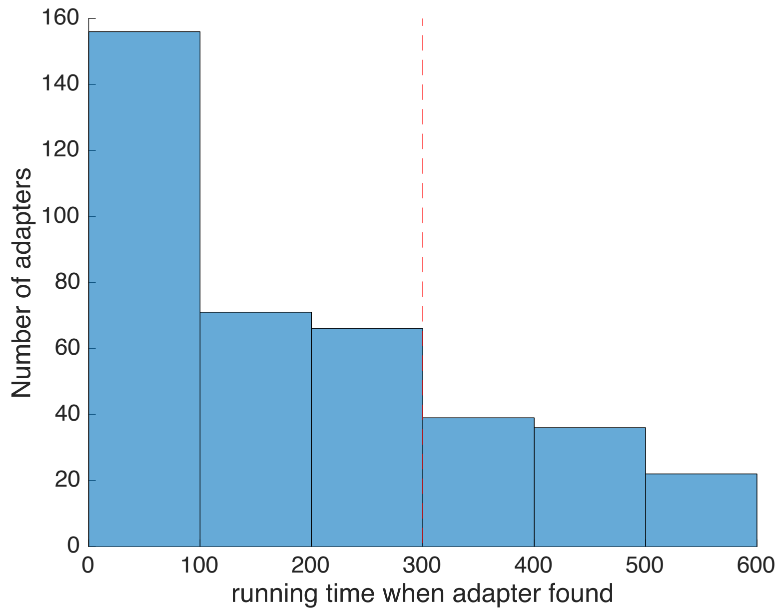


Figure 1.11: Running times for synthesized adapters using `median` reference function

modulo failures such as null dereferences. Using a tool like Daikon [20] 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 [21] 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 [14, 15] 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 [22], though we are not aware of any existing binary-level implementations that would be suitable.

1.6 Related Work

1.6.1 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 [23, 24, 25]. Jiang et al. [26] 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. Ramos et al. [11] 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. [27] 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. [28] 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. [29] 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 [30]. 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 [21]. More recently, Katz et al. [31] 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.

1.6.2 Component Retrieval

Component retrieval is a technique [32], [33], [34] 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 [35] and signature matching [36] 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

1.6.3 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 [37], for component adaptation using formal specifications of components [38], [39], [40], [41], [42]. Component adaptation has also been performed at the Java bytecode level [43], as well as at low-level C code [44]. Behavior sampling [45] 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 [37].

1.6.4 Program Synthesis

Program synthesis is an active area of research that has many applications including generating optimal instruction sequences [46, 47], automating repetitive programming, filling in low-level program details after programmer intent has been expressed [3], and even binary diversification [48]. Programs can be synthesized from formal specifications [49], simpler (likely less efficient) programs that have the desired behavior [46, 3, 47], or input/output oracles [50]. We take the second approach to specification, treating existing functions as specifications when synthesizing adapter functions.

1.7 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 2

Using Path-merging With Adapter Synthesis

Chapter 3

Path Merging With Symbolic Execution Of Java Bytecode

Chapter 4

Program Repair Using Adapter Synthesis

Chapter 5

Conclusion and Discussion

References

[1] 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.

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

[3] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay 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.

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

[5] Kesha Hietala. Detecting behaviorally equivalent functions via symbolic execution. 2016.

[6] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros 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.

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

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

[9] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek 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.

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

[11] David A Ramos and Dawson 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.

[12] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. `http://www.x86-64.org/documentation/abi.pdf`, 2013.

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

[14] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 193–204, June 2012.

[15] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *36th International Conference on Software Engineering (ICSE)*, pages 1083–1094, June 2014.

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

[17] Patrice Godefroid and Daniel 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.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim 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.

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

[20] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[21] Bruno Dutertre. Yices2.2. 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.

[22] Saurabh Srivastava and Sumit 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.

[23] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *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.

[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane 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.

[26] Lingxiao Jiang and Zhendong 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.

[27] 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.

[28] Yaniv David, Nimrod Partush, and Eran 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.

[29] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo 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.

[30] Adrià Gascón, Pramod Subramanyan, Bruno Dutertre, Ashish Tiwari, Dejan Jovanović, and Sharad 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.

[31] Deborah Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *Software Analysis, Evolution and Reengineering (SANER), 2018*. IEEE, 2018.

[32] Mikael 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.

[33] Colin Runciman and Ian 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.

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

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

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

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

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

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

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

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

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

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

[44] Marius Nita and Dan Grossman. Automatic Transformation of Bit-Level C Code to Support Multiple Equivalent Data Layouts. In Laurie 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.

[45] Andy Podgurski and Lynn 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.

[46] Henry 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.

[47] Rajeev Joshi, Greg Nelson, and Keith 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.

[48] Matthias Jacob, Mariusz H. Jakubowski, Prasad Naldurg, Chit Wei (Nick) Saw, and Ramarathnam Venkatesan. The superdiversifier: Peephole individualization for software protection. In Kanta Matsuura and Eiichiro 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.

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

[50] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish 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.