Finding Semantically Equivalent Binary Code By Synthesizing Adaptors

Kesha Hietala[1], Vaibhav Sharma[2], and Stephen McCamant[2]

[1] University of Maryland, College Park [2] University of Minnesota, Twin Cities

Overview

Our goal is to discover functions with the same behavior despite differences in their interfaces. To this end, we present a technique we call *adaptor synthesis* that determines whether the behavior of one function can be made to match the behavior of another function by appropriately modifying its arguments. Consider functions f1 and f2 below:

```
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);
}
```

For any integer x and unsigned integer y, f2(y,1,x,x) will return the same value as f1(x,y). We call the function that maps (x,y) to (y,1,x,x) an *adaptor.* With this particular adaptor, we can consider f2 to be *semantically equivalent* to f1. We write this relationship as f1 ← f2.

Applications

Security

If we allow for semantic equivalence between functions that have different error behaviors, then we can use adaptor synthesis to find different versions of a function with and without certain bugs. Adaptor synthesis can also be used to find different versions of a function with other desirable properties, such as efficiency or clarity.

Example: Adaptor synthesis can find that a call to OpenSSL's *BN_hex2bn* function, which has a null dereference/heap corruption bug (CVE-2016-0797), can be replaced with a call to mbedTLS's *mbedtls_mpi_read_string*.

```
long wrapped_BN_hex2bn(BIGNUM *h, int len);
                                   16
long wrapped_mbedtls_mpi_read_string(BIGNUM *h, int radix, int len);
```

Library Compatibility

Adaptor synthesis can ease the transition between different libraries by (1) making sure that the new library functions have equivalent behavior to the old library functions and (2) discovering necessary changes to function argument structures.

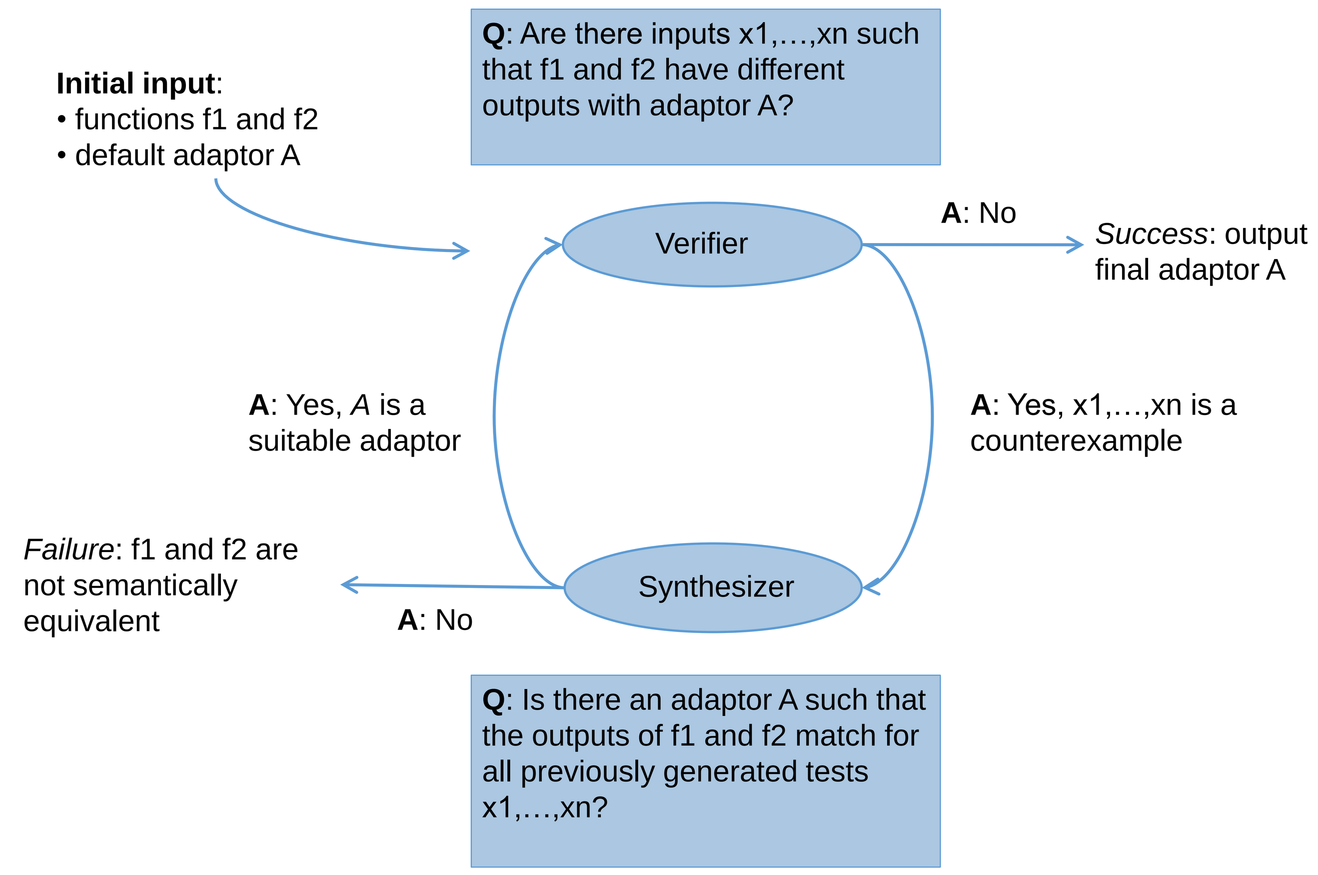
Example: Adaptor synthesis can help the programmer figure out how to replace mbedTLS's RC4 setup function with the RC4 setup function in OpenSSL.

```
typedef struct {                  typedef struct rc4_ket_st {
    int x;                            unsigned int x, y;
    int y;      8-to-64 zero extend   unsigned int data[256];
    unsigned char m[256];         } RC4_KEY;
} mbedtls_arc4_context;
```

```
void mbedtls_arc4_setup
  (mbedtls_rc4_context *ctx, const unsigned char *key, unsigned int keylen);
                         1
void RC4_set_key(RC4_KEY *key, int len, const unsigned char *data);
```

Algorithm

We use *counterexample guided inductive synthesis* (CEGIS) to search for an adaptor that maps the arguments of f1 to the arguments of f2 (and the return value of f2 to the return value of f1) in such a way that the behavior of the two functions match. Our specification for synthesis is the behavior of f1 and we define counterexamples to be inputs on which the behavior of f1 and f2 differ with a given adaptor.



The CEGIS search is restricted to a finite family of adaptors. One family of adaptors we support allows for an argument of f2 to be replaced by (1) an argument of f1, (2) a constant value, or (3) a type conversion applied to an argument of f1.

We have also experimented with adaptors that can replace arguments with the string length of a pointer argument or a bounded depth arithmetic expression and adaptors that can convert between different struct arguments. We also support simple adaptations of return values.

Implementation

We implement adaptor synthesis for Linux/x86-64 binaries using the symbolic execution tool FuzzBALL. We implement the CEGIS synthesizer and verifier loop by repeatedly executing the test harness below, alternating which variables are marked as symbolic.

```
void compare(x1, …, xn) {
    r1 = f1(x1, …, xn);
    y1, ..., ym = adapt(A, x1, …, xn);
    r2 = adapt(R, f2(y1, …, ym));
    if (r1 == r2) printf("Match\n");
    else printf("Mismatch\n");
}
```

• To find a counterexample we mark x1, …, xn as symbolic and look for paths that execute the "Mismatch" side of the branch.

• To find an adaptor we mark A (and R) as symbolic and and look for paths that execute the "Match" side of the branch.

• In addition to checking the return values r1 and r2, we also check that f1 and f2 make identical system calls and writes to memory.

Adaptors are represented using symbolic variables. The exact representation depends on the adaptor family being used, but as an example consider the case where arguments may be replaced by other arguments or constant values. Then you might associate two symbolic variables with each argument of f2: one that indicates what type of replacement will occur and one that indicates the replacing value.

Evaluation

As a large-scale evaluation, we ran our adaptor synthesis tool on 13,130 function pairs from the system C library (eglibc 2.19). Using a family of adaptors allowing argument substitution and type conversion, we found **8909** pairs to be inequivalent and **383** pairs to be equivalent. We also had **2989** timeouts and **849** crashes.

• Of the 383 equivalent pairs we found 28 interesting true positives, which are shown on the right.

• In the table, f1 ↔ f2 is shorthand for f1 ← f2 and f2 ← f1. # followed by a number indicates argument substitution, while the other numbers refer to constants. X-to-YS represents taking the low X bits and sign extending to Y bits, X-to-YZ is the same operation using zero extension.

• Sources of uninteresting true positives included unimplemented system calls in the C library (which write a value to errno and return -1) and functions that do nothing apart from returning a constant value.

• For scalability, we used a two minute hard timeout for adaptor synthesis (on a machine with 64GB RAM and an Intel Xeon E5-2680v3 processor), a five second SMT solver timeout, and limited the maximum number of times any instruction could be executed to 4000.

• The most common causes of crashing were missing system call support in FuzzBALL and incorrect null dereferences caused by improper initialization of pointer arguments.

f1 ← f2 or f1 ↔ f2 f(k) = f takes k args	adaptor
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(1) 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

Conclusions and Future Work

Our results confirm that several instances of adaptably equivalent binary functions exist in real-world code, and suggest that these functions can be used to construct cleaner, less buggy, more efficient programs. Some ideas for future work include:

• automatically generate binary code for adaptor functions
• experiment with other symbolic representations of adaptors
• add support for additional adaptor families (e.g. floating point values)
• infer preconditions in order to find adaptors that make functions equivalent provided that their preconditions are satisfied

Acknowledgements

This research was completed, in part, with support from the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-15-C-0110 and, in part, with support from the National Science Foundation under grant 1563920. We acknowledge the Minnesota Supercomputing Institute (MSI) at the University of Minnesota for providing computing resources for large scale evaluation experiments.