binCynth: Synthesizing C Programs From a Binary

Paul Pok Hym Ng[a]
[a]CS, Purdue University, ng121@purdue.edu
Nitin John Raj[a]
[a]CS, Purdue University, nitinjohnraj@gmail.com

Abstract

It is often the case where we want to understand how a black box binary works. This is especially true when dealing with legacy software that is no longer supported due to either the dissolution of a company, a 3rd party product with no source code available, or discontinued. However, such systems may be too deeply integrated into a framework making it hard or impossible to replace with a new system if a critical bug is discovered. Therefore we need a method to allow a software engineer to easily patch it.

To this end, we build a system to trace the execution of a binary, extract I/O of each black box function called, and synthesize each component function in C. These will then be composed to create a semantically equivalent program to the black boxed binary. As C code, this can then be easily modified and updated by any competent C programmer without the aid of binary instrumentation experts.

binCynth is simple and only requires a user to provide a comprehensive set of input examples and write a simple `main` function wrapper to the desired binary API call. Our tool will handle the rest.

Keywords: Program Synthesis, Formal Methods, Symbolic Execution, Disassembly, Decompilation, Binary Instrumentation

1 Introduction

In our world, many companies and institutions still rely heavily on legacy software that is no longer supported either due to company dissolution or product discontinuation. It is often normal for the computers running legacy software to be heavily isolated and cut off from external networks to prevent malicious entities from tampering and exploiting such software. Such systems also often rely on *security by obscurity.* However, such a solution is incomplete and unsafe. If there were other vulnerabilities in adjacent infrastructure, it is conceivable that an attack could be crafted to bypass multiple systems in order to exploit the legacy program.

Such intrusions are a very real thing. In 2021 an oil company, Colonial, was subject to legacy software attacks that resulted in a 5 million dollar ransomware attack. Another case of such an attack also occurred in 2021 which targeted Flagstar Bank. The attack on Flagstar bank took advantage of a now obsolete file transfer application (FTA) from Accellion.

In order to prevent this, an ideal solution would be to move towards a newer system that provides the inherent security benefits of modern software security advancements. Unfortunately, this is often too costly for many companies to consider and they choose to maintain the status quo and patch holes in outer layers of security.

Another approach to solving this problem would be to patch the existing legacy code and harden it to prevent attacks. However, as many systems used by companies are 3rd party systems, these institutions do not have access to the source code. As a result, to patch these binaries, one would need to resort to binary instrumentation. This is the act of injecting (either statically [6] or dynamically [7]) additional assembly into the binary to change the execution of the binary. Dynamic binary instrumentation provides a more robust and accurate method of modifying binaries but incur a significant performance overhead due to the fact that the instrumentation tool needs to actively track the execution and determine when to execute additional instructions. Static instrumentation incurs much less overhead (determined by the behavior of additional assembly added) as we do not need to monitor the binary during execution. Unfortunately, this has its own downsides. Firstly, after injection of additional assembly instructions, this will change the `jmp` offsets for instructions. This means that control flow instructions' target need to be recomputed. Secondly, static analysis may be inaccurate when analyzing obfuscated binaries. Obfuscation could have been a choice by the original software engineer or a result of compiler optimization.

Take the following binary bytes which represent the correct execution.

```
eb ff: jmp short 0001
26 05 00 03: add ax, 300h
00 ...: something
```

If however the static analysis began analysis at an additional one byte offset we would obtain a different set of disassembled instructions.

```
ff 26 05 00: jmp word ptr [5]
03 00: add ax, word ptr [bx+si]
```

This is because both data and code are (and should be) treated the same way in a binary.

In light of this, we would like to combine program synthesis together with symbolic execution to achieve what both static and dynamic binary instrumentation cannot. We will be not only be able to examine the behavior of the program with dynamic analysis tools but also synthesize semantically equivalent human readable C code to be patched. This C code then can then be compiled and run at native speed.

2 Related Work

In this section we will give a brief overview of what sort of tools and work exist which represent relevant components in our system.

2.1 C Program Synthesis

There exists a project known as Kalashinikov [3]. This synthesizer is one that takes in a formal definition of the target program to be synthesized (as a subset of C, C-). As its output it will return a C program.

Unfortunately, it can only synthesize a fragment of the C language. This is due to several things. Namely, its use of a fragment of second order logic and the fact that it targets finite domains.

However, they are still able to synthesize programs which are applicable for superoptimization, refactoring, and controller synthesis.

As such a synthesizer exists, we know that performing synthesis of C programs is definitely a (or at least partially) solvable problem.

2.2 Component Based Synthesis via Programming By Example

Strata [5] is a project which aims to retrieve the semantics of x86 instructions to be used for formal analysis via component based synthesis.

As a beginning set of components, they use and hand verified a set of 51 base assembly instructions and were able to retrieve the semantics of 1,795 out of the 2,918 instruction variants in scope (Haswell has a total of 3,684 instruction variants). Additionally, they were able to find specification bugs not only in the Intel x86 Manual but also in previous attempts (namely in Stoke [9]).

We ourselves are interested in using component based synthesis to synthesize functions from a binary. While Strata [5] does not attempt to synthesize C programs, it is still proof that such an idea can be applied at even the lowest level of execution.

2.3 Binary Instrumentation

This section discusses the two flavors of binary instrumentation.

2.3.1 Static Binary Instrumentation

Static binary instrumentation projects such as PEBIL [6] will insert new instructions into a binary. This will result in a new binary which can then be run as normal without any external tools. However such an approach encounters problems related to control flow. More specifically, if we have a jump instruction, `jmp 0x1234`, which sets the instruction pointer to execute code at address `0x1234` next, injecting assembly would require the re-calculation of such offsets. In the case of static address offsets, it is relatively simple to recalculate by offsetting static addresses by the number of additional bytes injected. However, instructions that overwrite the instruction pointer based upon registers or values in memory will require additional control flow analysis.

Additionally, static analysis runs into the problem mentioned above where static analysis may lead to the incorrect disassembly in complex instruction set computer (CISC) architectures such as x86. Such architectures have variable length instructions. In reduced instruction set computers (RISC) with fixed instruction size, this is less of a problem as we always know how far to advance the analysis at each step.

2.3.2 Dynamic Binary Instrumentation

Dynamic binary instrumentation tools such as Pin [7] inject assembly code as a binary runs. Such an approach will be able to circumvent static analysis pitfalls such as incorrect disassembly. This approach requires that the tool keep track of execution and inject assembly when certain instructions or conditions are met. The upside of this approach is that one does not need to modify the original binary. However execution time will increase anywhere from 2 times to 20 times [7]. This is highly undesirable and such performance impact is often unacceptable outside of an academic setting.

3 Contributions

We provide a system that has the following features.

- I/O example generation via CPU emulation
- Binary function splitting and obtaining I/O examples of internal functions
- Component based C synthesizer using programming by example
- Equivalence checking of synthesized programs
- A complete system combining the advantages of dynamic binary analysis and binary instrumentation without needing to handle assembly code

Each of these components will be discussed in the following sections. We will detail what tools we chose and why we chose them along with the details of how each component of our system was implemented.

Our system is simple and only requires the user to write a simple C `main` wrapper and provide a set of inputs to a target binary function to be synthesized. Suppose we had a library to be imported and had a function with two input arguments `libAPICall(a, b)`.

Such an input set would look like the following.

```
int32,52,int32,86,
int32,27,int32,95,
...
```

Such a `main` wrapper would look like the following.

```
#include "lib.h"
int main(){
  int a;
  int b;
  return libAPICall(a, b);
}
```

binCynth will then output synthesized functions in its components file.

4 Implementation

In this section we detail our binCynth implementation.

4.1 CPU Emulation

To be able to perform a dynamic analysis of the binary, we will be utilizing CPU emulators in order to get an accurate idea of what actually happens during execution. There are multiple different emulators which we could use such as QEMU [2] or Unicorn [8]. However, we chose Triton [1] (angr [10] also supports symbolic execution but we did not use it as we managed to get Triton working first). It was chosen as our emulator for two reasons. Firstly, it supports symbolic execution of binaries. This fact is used later for binary equivalence checking. Secondly, this way we would need to use one less tool in our tool-chain.

Triton however does not enable us to take advantage of fuzzing. We require the the user to provide their own set of inputs. While we did attempt to set up and use a tool known as AFLplusplus [4] which combines AFL, a fuzzer, with Unicorn, a CPU emulator, we were unable to easily obtain the required output information during emulation. This means that we are unable to take advantage of mutation based input generation to increase the input sample size.

To load the binary into our CPU emulator we use a tool known as LIEF [11] (Library to Instrument Executable Formats). This library provides us a very nice way of locating the `main` symbol, which we ensure exists via the wrapper, in a binary as the first entry point of execution.

4.2 I/O Example Generation

We assume that a user will know what inputs the binary's library API calls look like and we require they provide a wrapper to call that function with the correct input types. This is done for a very specific reason. Binaries may be compiled without any proper symbols delineating where functions begin and end. As such, by forcing a `main` function in the wrapper we will be able to leverage the fact that the user themselves inserted a `main` symbol in order to determine where to begin execution.

4.2.1 Setting up Input Arguments to the Top Level Binary Call

Currently we only support x86 binaries of the ELF format which use the System-V calling convention. Let us take a look at an example of a `main` function and how it sets up its call stack for a function, `f`, to be called.

Suppose we have the following wrapper.

```
int main(){
    int a;
    return f(a);
}
```

The disassembly of that main function is as follows.

```
000000000000113c <main>:
    113c: endbr64
    1140: push rbp
    1141: mov rbp,rsp
    1144: sub rsp,0x10
    1148:  mov eax,DWORD PTR [rbp-0x4]
    114b: mov edi,eax
    114d: call 1129 <f>
    1152: leave
    1153: ret
```

The instructions we would like to focus on are `1140, 1141, 1144, 1148` and `114b`. `1140, 1141, 1144` represent the setup of `main`'s stack by allocating space for local variables, updating `rbp` (base stack pointer) and `rsp` (stack pointer). In `1144` notice that we subtract `0x10` from `rsp` telling us that `0x10` bytes have been allocated for `main`'s stack. In `1148` the first argument to `f` is moved into the register `eax` before it is finally moved into `edi` prior to the call to `f`. The reason why `114b` exists is due to the System-V x86 calling convention. Integer input arguments to functions are placed in the registers in the following order `RDI, RSI, RDX, RCX, R8` and `R9`.

By utilizing this standard calling convention and our fixed wrapper, we are then able to consistently overwrite the values offset from `rbp` by emulating the first 4 instructions of our wrapper to determine the size of the stack and the value of `rbp`. Finally, the offset from `rbp` (in this case 4 as signed integers are 4 bytes large on x86-64), is determined by the size of the input argument. This is something which we assume the user knows and we require this information.

In the table below, we show the stack with 0 representing the initial call from `_start` to `main`. Below we used a 64bit argument to `f` hence the -8 in the address.

Offset	Value
...	...
-40	f's Caller RBP (main)
-32	f's Caller Ret Addr (main)
-24	First Arg to f
-16	Main's Local Variable
-8	Main's Caller RBP (_start)
0	Main's Caller Ret Addr (_start)

To obtain output, we once again take advantage of the x86 System-V calling convention. In our case we look for `eax` and `rax`. These are the integer return values of functions. Therefore, we just need to obtain the value in these registers when `main` completes execution.

4.2.2 Extracting I/O From Internal Calls

This is required as a library function `libAPICall()` may call another library function `libAPICallInternal()`. In this case we need to take into account what this would look like. In general, the instructions executed will be as follows

```
; function libAPICall execution
...
mov edi, eax
call <libAPICallInternal>
; begin libAPICallInternal
push rbp
mov rbp,rsp
sub rsp,0x??
; Retrieve input arguments
mov DWORD PTR [rbp-0x4],edi
mov DWORD PTR [rbp-0x8],esi
...
leave
; libAPICallInternal Completes
ret
; return to libAPICall
```

As mentioned in the section above we know a function completes when we hit a `ret` therefore we obtain the return value from there. We also know that arguments are stored with offsets on the stack. Therefore we can also retrieve them from there. Specifically, we will use Triton's ability to record what instructions read from specific addresses. If an instruction writes `rbp - offset` from any of the input argument registers, then we know that this is an input argument. Additionally, we can determine the size of the input argument by the register that it was retrieved from (recall `exx` is 32 bit and `rxx` is 64 bit).

4.3 Component Based Synthesis Engine

In this section we detail the internals of our synthesis engine.

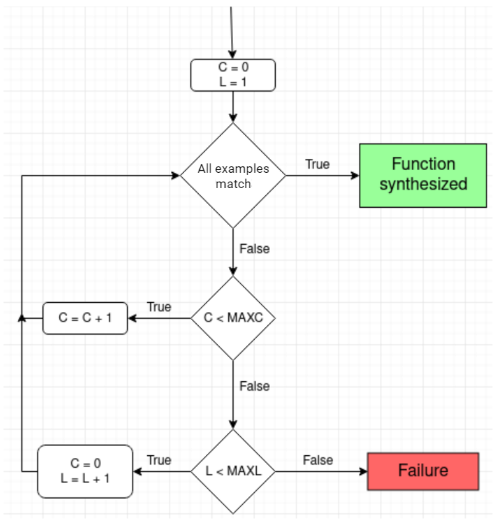
4.3.1 Language Choice

We chose to write our synthesis engine in C++. The choice of language was solely driven by the target synthesis language C/C++. By writing our synthesis engine in C++ we are able to leverage the fact that any execution of our components will be behaviorally equivalent to the final synthesized function when compiled. This also means we can take advantage of function pointers.

4.3.2 Parametrization

We follow the Occam's Razor approach when it comes to our synthesized program. This means we wish to synthesize programs that are as short as possible and contain a minimal number of constants. We will use a parametrization approach similar to Kalashnikov [3]. Specifically we will parametrize over both the number of constants c and the length of our program (number of components) l.

Below is a flow chart of how we modify these two parameters during the synthesis process.



We will always begin synthesis by only using 1 component and 0 constants. We will then check if a specific permutation of components (and the input arguments chosen) match all I/O examples. If it does we will terminate and return the synthesized function. Otherwise we will check if we can increase the number of constants $c < MAXC$. If we have reached the max number of constants we then attempt to increase the number of components and reset the number of constants to 0. In the case where we have reached the max number of components we wish to use $l = MAXL$, we terminate with a fail condition.

$MAXC$ is determined by the components that we are using. If we are using 4 components, three of which have 2 input arguments and one which has 1 input argument, we have $MAXC = 3 \cdot 2 + 1 \cdot 1$. To constrain the number of values we need to test for each constant input, we have a user defined max bit size of a constant. In our testing, this is set to 4 allowing for constants between $-2^4 \leq constval \leq 2^4 - 1$. $MAXL$ is also a user provided parameter. In our testing we set $MAXL = 5$.

4.3.3 Component Input/Output Variables

We synthesize programs in Static Single Assignment (SSA) form. As a result, a component's input variables is dependent on its location in the program. While every component can always take constants and root input variables as input, it also needs to be able to take the output of a previously executed component.

For example take the following program.

```
int comp1(int a){... return outN;}
int comp2(int a, int b){... return outN;}
int synthed(int a){
    int out1 = comp1(??);
    int out2 = comp2(??, ??);
    return out??;
}
```

In the synthesized function, `comp2` could take `int a`, constants, or `out1` as input. However `comp1` is forbidden from taking future output arguments `out1` and `out2` as input.

With regards to return value, we will always return an output from a component. This does not mean we always need to choose the *last component output*. In the case of `synthed`, it could either return `out1` or `out2`.

4.3.4 Component Implementation

As mentioned previously we use function pointers to implement our components. This gives us the advantage of storing all components in a list and generate permutations of components used as indices into this `FUNCS` array.

To ensure that we have the correct function definition when executing a component we define additional `typedef`s.

```
typedef void
    (* func_ptr) (void);
typedef int
    (* func_ptr_o_int_i_int)(int);
typedef int
    (* func_ptr_o_int_i_int_i_int)(int, int);
```

`func_ptr` is the base type which can then be typecasted to any of our additional function prototypes. As one can clearly see we use a format of

- func_ptr
- output type
- input argument type

This then allows us to store these as follows without the compiler complaining.

```
const func_ptr FUNCS[FUNCS_NUM] {
    (func_ptr) int_add,
    (func_ptr) int_sub,
    (func_ptr) int_mul,
    ...
};
```

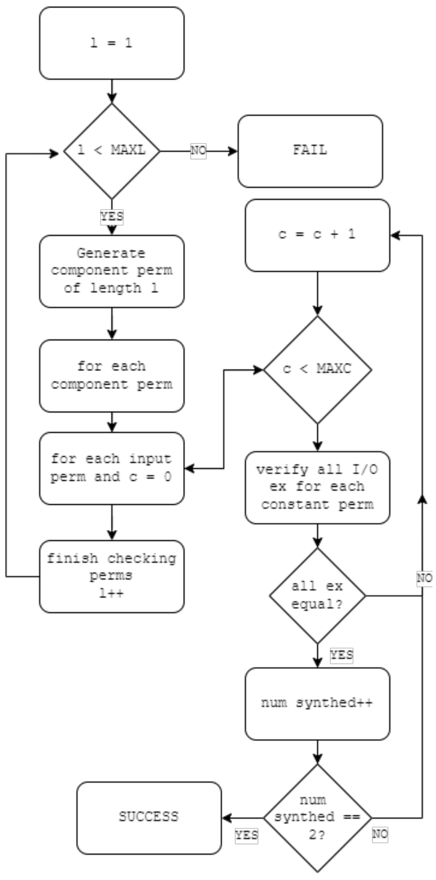
4.3.5 Function Synthesis Order

We will always synthesize functions in *reverse call order*. Suppose we have a function `f` which calls `f1` and `f1` calls `f2`. The synthesizer will then synthesize `f2` before `f1` and `f1` before `f`. This ordering is determined by the function I/O extraction procedure and the synthesis engine is passed I/O examples in the reverse call order.

This is done because we know that in the binary these dependencies exist. However, this does not mean that our synthesizer will produce a `synth_f` with the same call order. It just ensures that if the dependency is discovered the synthesizer will be able to use it.

4.3.6 Synthesis Loop

We now provide a high level description of how the synthesis loop works. Below is a flow chart of it.



l = 1
l < MAXL
NO
FAIL
YES
Generate component perm of length l
for each component perm
for each input perm and c = 0
finish checking perms l++
c = c + 1
c < MAXC
verify all I/O ex for each constant perm
all ex equal?
NO
YES
num synthed++
num synthed == 2?
SUCCESS
YES
NO

We begin synthesizing with $l = 1$. We then proceed to generate all the permutations of programs which use l components (all orderings of $l = 3$ eg. 000, ..., 012, 120, 201, ...). Note that these permutations additionally take into account orderings of valid input arguments. We then proceed to check if each permutation produces equivalent output for all I/O examples provided. If it does then we increment the number of synthesized programs. In the case where we have reached are target number of synthesized programs (default 2) we terminate with a success and return both synthesized functions.

In the case where we reach $MAXL$ without synthesizing our target number of functions, we terminate with a failure.

We choose to synthesize two functions that satisfy the same set of I/O examples for the following reason. In the case that the input example set is not comprehensive enough, it may be possible to synthesize semantically different functions that just so happen to pass all examples.

```
int synth_1(int a, int b){return 0;}
int synth_2(int a, int b){return a * b;}
```

The above functions are obviously semantically different however if we were to only provide the input examples,

- Out: 0, In: 0, 0
- Out: 0, In: 5, 0
- Out: 0, In: 0, 10

both would satisfy the output values.

The fact that these functions are semantically different will be discovered in the equivalence check and we will need to re-attempt synthesis after generating a more comprehensive example set.

5 Equivalence Checking

Synthesis of functions from the binary black box is done from a set of base components. However, once we find a function f that satisfies the I/O examples for the given binary, we must synthesize another function g which is behaviorally equivalent to f. This is to prevent overfitting on the I/O examples.

We use Triton as a symbolic execution engine to help check function equivalence. We start with the functions f and g, and augment each of them with a simple `main` function that declares (but not initializes) variables fed as arguments to the function and returns the result of a call to f or g. We use Lief to parse the binary.

5.1 Undecidability of Functional Extensionality

There are two main sources of undecidability that we must consider:

1. Unbounded types
2. Unbounded recursion

We choose to avoid explicitly addressing these problems by considering programs within a decidable fragment. The former is avoided since we only consider types which are bounded (integers and long integers). Moreover, if we were to additionally consider memory as discussed in Section 7, we will also place bounds on the amount of memory which could be statically or dynamically allocated. The latter is circumvented by restricting our base components, preventing us from having programs with recursion. These choices limit the kinds of programs that we can synthesize. Extending our work to consider more complex programs is future work.

5.2 Symbolic Variables

We symbolize the registers and memory using Triton. Symbolizing the memory within Triton is important, because without explicitly doing so Triton sets memory to contain only 0s. This assumption leads to incorrect inferences, like the equality of the functions like the example below below.

```
int synth_1(int a, int b) {return 0;}
int synth_2(int a, int b) {return a;}
```

`a` and b (which are arguments in stack memory) are equal to the concrete value 0 if we do not explicitly symbolize the memory.

Symbolization of registers is done with the `TritonContext.symbolizeRegister` function. Symbolization of the memory containing the arguments is done by constructing `MemoryAccess` objects, and calling the `TritonContext.symbolizeMemory` function. We additionally use `(declare-fun)` in Z3 to create symbolic BitVectors which generalize over any value the symbolic variables may hold.

5.3 Obtaining Constraints From Triton

We first compile the functions augmented with the fixed format of our `main` function. We use Lief to parse the binaries and load the code into memory. Triton then begins execution at the `main` symbol. For each assembly instruction parsed, we create a `Instruction` object as input to a `TritonContext` to execute and build internal abstract syntax trees and constraints. By calling `Instruction.getSymbolicExpressions` on each assembly instruction executed, we retrieve SMT constraints for our binaries, representing the CPU execution behavior. We add prefixes to every symbolic function thus allowing us to distinguish between similarly named references in both binaries.

5.4 Equivalence Assertion

Once we have the constraints for each binary, we must check the validity of the following assertion: If the binaries start with the same initial register state and the same initial memory state (which includes the arguments passed to their respective functions), then their return values are equal.

Checking the equality of the registers and memory is done using the symbolic variables declared in Section 5.2. It is important to assert the equality of the initial register state as a different value in a register such as ZF (zero flag) may result in different control flow behavior on conditional jumps, such as the `jz` (jump if zero flag is set) instruction.

```
(define-fun init-cond () Bool (and
    (= _0_eax_init _1_eax_init)
    ...
    (= _0_mem_loc_0 _1_mem_loc_0)
    ...
    (= _0_mem_loc_N _1_mem_loc_N)
    ...
))
```

Checking the equality of return values is done by asserting the equality of the final state of the `eax` return register. The final assertion is given below. We additionally negate this implication to have Z3 return `unsat` for validity instead of a satisfying example.

```
(assert (not (=> init-cond
  (= _0_final_eax _1_final_eax))))
```

Essentially, the equivalence of the initial state implies that the final result of both binaries must be equal.

6 Evaluation

We present two examples which demonstrate how binCynth performs. Both tests were run on the following system.

- CPU: Ryzen 3900x
- Memory: 64GB DDR4 RAM at 3200MHz
- OS: Ubuntu 20.04 LTS on WSL2 (Windows Subsystem for Linux)

Each example was synthesized with 100 input examples to the binary and the same set of base components. If there is more than one function to be synthesized, each component will be added to the base component set before additional functions are synthesized.

6.1 Example 1

Suppose we had the following binary function `f` which takes a single integer argument and returns an integer argument. This is then wrapped in a `main` wrapper and compiled.

```
int f(int a) {return a - 15;}
int main() {int a; return f(a);}
```

The resulting function synthesized is shown below.

```
int synthed_5(int in_1) {
        int out_0 = int_add(-15, in_1);
        return out_0;
}
```

Note that the synthesized function does not actually use the subtraction component but rather the addition one with -15 as a constant. This is just because the synthesizer tried addition before subtraction.

Below are the run time statistics.

Time	Task
0.662 sec	Binary I/O Extraction
0.0321 sec	Function Synthesis
0.0684 sec	Function Equivalence

6.2 Example 2

This is a more complicated example with nested function calls and variable updating. Suppose we had the input binary function `f1` which in turn uses `f2` and `f3`.

```
int f3(int a, int b){
        return 2 * a - b;
}
int f2(int a, int b){
    return a + f3(a, b);
}
int f1(int a, int b){
    a = a + 1;
    return a + f2(a, b);
}
int main(){
    int a; int b;
    return f1(a, b);
}
```

We are then able to synthesize

```
int synthed_f3(int in_1,int in_2){
        int out_0 = int_sub(in_1, in_2);
        int out_1 = int_add(in_1, out_0);
        return out_1;
}
int synthed_f2(int in_1,int in_2){
        int out_0 = int_sub(in_2, in_1);
        int out_1 = synthed_f3(in_1, out_0);
        return out_1;
}
int synthed_f1(int in_1,int in_2){
        int out_0 = synthed_f3(in_1, -2);
        int out_1 = synthed_f3(out_0, in_2);
        return out_1;
}
```

This is an excellent display of what was discussed previously in the synthesis engine section. The final synthesized `f1` does not actually make use of the synthesized `f2` at all, instead it makes two different calls to the synthesized `f3` to create a semantically equivalent `f1`.

Below are the run time statistics.

Time	Task
1.2714 sec	Binary I/O Extraction
1.2435 sec	`f3`: Function Synthesis
2.8948 sec	`f2`: Function Synthesis
8.2539 sec	`f1`: Function Synthesis
0.0839 sec	`f3`: Function Equivalence
0.0902 sec	`f2`: Function Equivalence
0.0923 sec	`f1`: Function Equivalence

The time taken to synthesize functions seems to increase exponentially with the number of functions synthesized, suggesting that our synthesis engine does not scale to larger programs. This is unsurprising, since our synthesis engine is simply a proof of concept and is naively implemented. Adding support for multi-threading and better heuristics for our synthesis engine is delegated to future work.

7 Limitations and Future Work

Our current implementation while complete for certain types of programs leaves much to be desired. We will now discuss some of the limitations and future work which will improve the viability of our system.

7.1 Target Architecture and Binary Format

We target x86-64 binaries compiled in an ELF format with the System-V calling convention. The CPU emulator we use, Triton [1], provides support for x86, x86-64, ARM32 and AArch64 architectures. We only target one. Moreover, LIEF [11] supports ELF, PE, and Mach-O format binaries and we only target ELF.

It would be simple to extend binCynth to support these additional architectures by rewriting small parts of the binary I/O extraction and equivalence checking. Specifically we would need to modify how we extract/overwrite input arguments and how we retrieve return values.

7.2 Standard C Library

Triton [1] supports hooking the C library into the CPU emulation. This means that C library functions such as `malloc, free, memset`, or `memcpy` can be emulated faithfully. As Triton also tracks memory reads and writes, we would also be able to extract the state of memory that was allocated, read, or written to (both concretely and symbolically).

With regards to the synthesis engine in order to provide support for C function calls, we would create additional components which wrap these C library calls.

7.3 Other Component Input/Output Types

Our current synthesis engine only supports integer programs. This is clearly a very important limitation to resolve. It is trivial to support additional types such as `long` but other types such as `float` require additional design considerations. We will discuss how we could resolve some of the `float` problems in the heuristics section.

With regards to memory, statically allocated arrays is a simple extension. We only need to create read and write components which take an array pointer, index, and value as input. In order to prevent our synthesis engine from seg faulting, we will only execute such calls if they do not read and write out of array bounds.

For dynamically allocated memory, we would need to additionally keep track of the pointers that were returned by calls to `malloc`. However, the rest of the implementation will be similar to how we do it for statically allocated arrays.

We will also need to restrict the size of the memory that is allocated. This is due to the fact that during symbolic execution we will need to symbolize any memory that was used. By allowing a large portion of memory to be symbolized we will run into performance issues.

7.4 Synthesis Heuristics and Strategies

As a simple home-brew component based enumerative synthesizer, we have close to 0 heuristics implemented. The only "heuristic" used is the aforementioned c and l parameters which we increment as the synthesis process proceeds.

An interesting synthesis strategy to implement would be genetic programming and incremental evolution. Such strategies offer a performance metric for measuring how well a synthesized function "fits" our I/O examples. This way we could hone in on a correct function faster.

Additionally, we believe that using genetic programming may provide a way to combat floating point calculations. While it is computationally infeasible to enumerate constants over all IEEE FP numbers in either a 32-bit or a 64-bit format we may be able to instead enumerate over integral divisions of FP numbers. For example, instead of incrementing the bits of a `double` (0x00000000, 0x00000001, ..., 0xFFFFFFFF), we could instead increment in user defined intervals such as 0.25. This means that we would enumerate over $-10.25, ..., -0.25, 0, 0.25, ..., 10.25$ and so on. By having a fitness metric we could determine which synthesized function is the "best". While this appears to be error prone, if the division size is small enough, the cascaded error between multiple FP component calls may be small enough for the correct program (within an error range ϵ) to still be synthesized. Just like Max-Flow algorithms we could use a scaling parameter to decrease interval size (such as dividing by 2) as fitness increases. To ensure termination, we could additionally use an error value ϵ to terminate execution if the output of examples match within a error range.

7.5 Performance

Currently, our synthesis engine is single-threaded and only checks one permutation of components one at a time. While it would require modifying the data structures a bit, it should be trivial to allow concurrent execution of component permuations. This is because each permutation executes independently of each other.

7.6 Base Component Set

Our component set is also nowhere near comprehensive. As a prototype we only included a subset of basic arithmetic operations. This is in contrast to Strata [5] which carefully evaluated the base set of components before running their component based synthesis engine. Consequently, we are unable to synthesize all integer programs, but binCynth is definitely capable of doing so if we were to add additional components.

7.7 Constant Lifting From Binary

Currently we enumerate over a constant range defined by a user set bit size. However there is additional information which could be taken advantage of.

In binaries, in addition to the `.text` section which stores code we also have the `.bss` and `.data` sections. Specifically, `.bss` stores uninitialized global variables and the `.data` contains initialized global variables (such as `#DEFINE`s). As a heuristic we could always try initialized global variables from `.data` before any other constants. Additionally, as Triton allows us to track memory writes/reads, we could also track the usage of variables in the `.bss` section during binary I/O extraction. We would then once again give these discovered constants a higher priority before our enumerated ones. Such constants gleamed from the binary may also be out of the range of our maximum constant bit size allowing us to explore a state space which we would otherwise be unable to.

Lastly, some programs may initialize local variables with a constant value (`int constant = 0x500;`). During execution we would be able to observe an instruction which would write the value 0x500 to the stack as a local variable before use. Such an instruction would take the form `mov DWORD PTR [rbp-0x8],0x500`. We would then easily be able to tell that the function has a local variable which at some point was initialized to the value 0x500. These would then once again be passed with higher priority than our enumerated constants.

8 Conclusion

binCynth is a complete tool which is able to synthesize C programs from a black boxed binary. While it has its limitations, we believe that many of the limitations discussed above can be resolved with more development time. As a prototype, we believe that binCynth has successfully demonstrated the use of program synthesis and PbE in the context of decompilation and binary instrumentation. By approaching the problem from a different view and mindset, we are able to subvert certain limitations of existing methodology such as compiler optimization flags. This is because we work strictly from I/O examples and do not need to reason directly about optimizations made.

References

[1] Triton: A Dynamic Symbolic Execution Framework, SSTIC, 2015.

[2] F. Bellard, Qemu, a fast and portable dynamic translator., in: USENIX annual technical conference, FREENIX Track, Vol. 41, Califor-nia, USA, 2005, p. 46.

[3] C. David, P. Kesseli, D. Kroening, M. Lewis, Program synthesis for program analysis, ACM Transactions on Programming Languages and Systems (TOPLAS) 40 (2) (2018) 1–45.

[4] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, Afl++: Combining incremental steps of fuzzing research, in: 14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20), 2020.

[5] S. Heule, E. Schkufza, R. Sharma, A. Aiken, Stratified synthesis: automatically learning the x86-64 instruction set, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2016, pp. 237–250.

[6] M. A. Laurenzano, M. M. Tikir, L. Carrington, A. Snavely, Pebil: Efficient static binary instrumentation for linux, in: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), IEEE, 2010, pp. 175–183.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, Acm sigplan notices 40 (6) (2005) 190–200.

[8] N. A. Quynh, D. H. Vu, Unicorn: Next generation cpu emulator framework, BlackHat USA 476.

[9] E. Schkufza, R. Sharma, A. Aiken, Stochastic superoptimization, ACM SIGARCH Computer Architecture News 41 (1) (2013) 305–316.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, Sok: (state of) the art of war: Offensive techniques in binary analysis.

[11] R. Thomas, Lief: Library to instrument executable formats (2017).