Case Study on LLVM as suitable intermediate language for binary analysis

Florian Märkl, *Technische Universität München*

Abstract

Many binary analysis tools and compilers, instead of directly working on code, use an intermediate representation of it. The idea of this thesis is to use the well-tested intermediate representation from LLVM for binary analysis tasks. We take a look at McSema, a tool to translate x86 and x86_64 binaries to LLVM, describe its translation process in detail and additionally implement Python bindings for it. To practically test McSema, we present five examples of code we translate to LLVM and then recompile again. The last of these demos is an example on using KLEE, a symbolic execution engine for LLVM, on the code produced by McSema in order to successfully solve a CrackMe. We conclude that McSema's translation approach provides a suitable way to extract functions from binaries to integrate them in other code or to analyse them using symbolic execution, as well as serving as a potential basis to implement an LLVM-based decompiler. We also compare it to Remill, another tool similar to McSema, which generates code that represents the assembly code more explicitly and VEX, the intermediate representation used in Valgind and Angr, which is also more close to the machine code.

1 Motivation

Many tools and frameworks exist to analyze binaries in a lot of ways. Especially when performing some sort of automatic analysis, most of these tools do not directly work on the raw machine code. Instead, one common approach is to first generate some sort of intermediate representation of the code, so the analysis algorithms do not have to care about aspects such as the original CPU architecture.

Similar techniques can be observed when looking at the architecture of common compilers. For instance, LLVM, which has a very modular architecture, contains a whole programming language called LLVM IR, which we will describe in detail in 2.3.1. Because many projects need to be able to rely on their compiler to always generate the right output for their input, compiler developers attach great importance to stability and correctness.

This thesis is based on the idea to use a robust and well tested intermediate language from a compiler in order to produce an intermediate representation for machine code instead of generating it from a high level language as usual.

2 Background

2.1 Intermediate Representations (IR)

An intermediate representation, henceforth referred to as IR, is a language or another form of data that is used to represent source code. IRs are commonly used internally in compilers and virtual machines and often have specific characteristics to suit the intended purpose.

2.2 Static Single Assignment (SSA)

Static Single Assignment (SSA) is a property of a language which adds the rule that every variable is assigned in exactly one location of the program. If a variable has to be written to multiple times, new versions of this variable are created.

2.3 LLVM

LLVM is a "collection of modular and reusable compiler and toolchain technologies", which roughly means, it is an architechture that allows building compilers to translate from an arbitary language to a variety of target architechtures. It has gained popularity over the last years and is, for example, used as the default compiler on macOS and as part of the compilation process for the relatively young language Swift.

The normal compilation process in LLVM can be coarsely divided into three different stages. The frontend takes the original code and translates it into LLVM's custom IR, called LLVM IR. When this representation has been obtained, several optimizations can be performed on the code. LLVM already provides powerful features for this purpose. The last stage, called the backend, is then responsible for translating the IR into native machine code.

Because of this modular architechture, compilers for new programming languages can be developed quite easily. Essentially, it is sufficient to create a frontend that translates the language to LLVM IR. This frontend does not necessarily need to do any optimization and can perform a very naive translation. This is why, nowadays, a huge variety of frontends for different programming languages exist, including C, C++, C#, Delphi, GLSL, Java, Lua and Python.

2.3.1 LLVM Intermediate Representation

LLVM IR is a language that is quite close to assembly language, but still independent of both the original language and the target architecture in a compilation process. Although in practice, LLVM IR is mostly used as an internal representation that the user of a compiler does not even need to know about, in theory, it can also be used to write complete programs in. Here is an example of a "Hello World" program, taken from the official language reference: [1]

```
; Declare the string constant as a global
    constant.
@.str = private unnamed_addr constant [13 x i8] c
    "hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {   ; i32()*
  ; Convert [13 x i8]* to i8  *...
  %cast210 = getelementptr [13 x i8], [13 x i8]*
      @.str, i64 0, i64 0

  ; Call puts function to write out the string to
      stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

LLVM IR is type safe and conforms to the Single Static Assignment form explained in 2.2. Code is organized in modules, which can be coursely described as the term for one file of LLVM code and roughly corresponds to one source file in C. LLVM code can be represented either in human-readable text form like the example above (`.ll` files) or encoded as bitcode (`.bc` files).

A module can contain functions, global variables, symbol table entries and metadata. The example consists of the global variable `.str`, a declaration of the external function `puts`, the function `main` and some metadata.

Functions in LLVM are very similar to functions in C. They can have any number of arguments and optionally a single return value.

Identifiers that start with a `@` sign describe a global symbol, while the ones starting with `%` are only local. Metadata definitions begin with a `!` sign.

LLVM provides, among others, basic integer types, such as `i32`, pointer types like `i32 *`, arrays and struct types which can be defined like this:

```
%somestructtype = type { i32, float, i32 * }
```

Code in functions consists of a series of instructions and labels which can be jumped to. Common control flow statements such as if/else or loops do not exist in LLVM and are instead modeled using branch instructions:

```
@str = private unnamed_addr constant [52 x i8] c"
    Second parameter is higher or parameters are
     equal.\00"
@str2 = private unnamed_addr constant [27 x i8] c
    "First parameter is higher.\00"

define void @nil_recurring(i32 %a, i32 %b) #0
{
entry:
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %if.then, label %if.else
```

```
9
10 if.then:
11   %puts2 = tail call i32 @puts(i8* getelementptr
         inbounds ([27 x i8]* @str2, i64 0, i64 0))
12   br label %if.end
13
14 if.else:
15   %puts = tail call i32 @puts(i8* getelementptr
         inbounds ([52 x i8]* @str, i64 0, i64 0))
16   br label %if.end
17
18 if.end:
19   ret void
20 }
21
22 declare i32 @puts(i8* nocapture readonly) #1
```

2.3.2 Passes

LLVM includes strong features for optimizing its IR code. These are implemented in the form of so-called passes which analyse or transform the code in some way. Examples for these passes are `-memdep`, which determines what operations a given memory operation depends on, or `-dce`, which performs dead code elimination.

Usually, these passes are executed by the user indirectly through either the compiler frontend, for example by specifying the `-O3` argument to clang, or the tool `opt` which optimizes given LLVM code.

2.4 McSema

McSema is a framework, developed by the security company Trail of Bits and available under a BSD license, which allows to translate native x86 and x86_64 code to LLVM IR. It takes a modular approach on the translation task by splitting it into two seperate parts: Control Flow Graph (CFG) recovery and the actual translation to LLVM IR. The translation approach is explained in detail in 4.1.

2.5 KLEE

KLEE is a symbolic execution engine built upon LLVM. Its primary purpose is to automatically generate test cases with a high code coverage for software projects. It operates directly on LLVM IR code, which would normally be compiled directly from the original source code in a high level language. [6]

3 Related Work

3.1 Valgrind and VEX

Valgrind is a framework for debugging and profiling programs on Linux. It is divided into a single core part and multiple tools for different purposes. As an example, Memcheck supervises all allocations, reads and writes of memory while running the program and detects, among other things, if any memory is accessed in a wrong way or the program has any memory leaks. Overall, Valgrind is a very mature project that contains production-quality tools and has been extensively tested.

Under the hood, Valgrind uses a custom IR called VEX. A program in VEX consists of multiple code blocks. These blocks have exactly one entry and can have multiple exits and consist of a list of statements, one or more jump and information about the types of temporary values used in the block [5].

Statements in VEX can be seen similar to instructions in machine code. They are executed and can alter the current state of the program. This state, called the guest state, is saved in a block of memory that contains the contents of registers and is accessed using `Get` and `Put` statements.

This is a short example of how x86 code can get translated to VEX, taken from the libvex source code:

```
addl %eax, %ebx
```

```
------ IMark(0x24F275, 7, 0) ------
t3 = GET:I32(0)      # get %eax, a 32-bit integer
t2 = GET:I32(12)     # get %ebx, a 32-bit integer
t1 = Add32(t3,t2)    # addl
PUT(0) = t1          # put %eax
```

The `IMark` contains information about the position and length of the original instruction, while `t1`, `t2` and `t3` are temporary variables.

3.2 Angr

Angr is a toolkit for binary analysis, available as a Python framework that provides a variety of functionality to load and analyze binaries instead of being a single program. This makes Angr relatively flexible to use, because the user can either access it using an interactive Python shell or write a small script. [9]

This is an example of a simple script that finds out the password a program is asking for by performing symbolic execution:

```
import angr
import base64
```

```
3
4 proj = angr.Project('./password')
5
6 argv1 = angr.claripy.BVS("argv1", 16 * 8)
7 initial_state = proj.factory.entry_state(
8         args=["./password", argv1])
9
10 initial_path = proj.factory.path(initial_state)
11 path_group =
12         proj.factory.path_group(initial_state)
13 path_group.explore(find=0x4005ce, avoid=0x4005df)
14 found = path_group.found[0]
15
16 pw = found.state.se.any_str(argv1)
17
18 print base64.b64encode(pw)
```

Internally, Angr uses VEX as its intermediate representation, which has been chosen, because reliable translation methods from many architectures already existed in Valgrind.

3.3 Remill

Remill is another framework to translate binaries to LLVM developed by the same company that developed McSema, Trail of Bits. It is similar to McSema in a lot of ways, for example it also seperates CFG recovery from translation, but it also takes different approaches in some ways. One big difference is that, while McSema translates whole functions from the binary as LLVM functions, Remill creates a single function for each basic block in the CFG. Its goal is primarily to generate an LLVM equivalent of the assembly code that explicitly represents the original control flow and memory usage. [3] [4]

4 Approach

This thesis focuses on the lifter McSema. We will describe in detail how it translates machine code to LLVM and then use it in a number of demos. All code mentioned here is available on the accompanying git repository [2].

4.1 McSema in-depth

In order to translate x86 or x86_64 to LLVM IR, McSema takes a modular approach by splitting the process into two seperate parts. First, a Control Flow Graph (CFG) has to be generated on the original file, which can be done in multiple ways as described in the following. To store the CFG, a custom file format based on Google Protobuf is provided by McSema. Taking this CFG, the program `mcsema-lift` can then generate the equivalent LLVM IR code. We will describe the translation process in detail later.

4.1.1 CFG recovery using bin_descend

One way to recover control flow used to be provided by McSema directly through the program `bin_descend` which takes a binary and generates a CFG through recursive descend.

Unfortunately, development of `bin_descend` has eventually fallen behind and it has been removed completely from McSema's official git repository during the time of writing.

4.1.2 CFG recovery using IDA Pro

Another way for CFG generation is using the proprietary program IDA Pro by Hex Rays. McSema provides an IDAPython script for this purpose that lets IDA Pro recover the control flow and then exports exactly what IDA Pro finds out to McSema's CFG file format. To conveniently use it, mcsema provides a tool called `mcsema-disass` that automatically starts IDA Pro and runs the script inside it.

Since IDA Pro's control flow recovery is fairly reliable, this approach is currently the most robust one and thus also the recommended way for CFG generation.

4.1.3 Translation to LLVM IR

After control flow has been recovered, `mcsema-lift` can translate it into LLVM IR. McSema's way of translating to LLVM is a very naive one. It does not attempt any optimization of the code during the translation step and thus produces a certain amount of unnecessary code in most cases. Instead, this resulting code is supposed to be passed through LLVM's optimizer, which can remove unneeded computations. This approach makes the implementation of the translation fairly easy since instructions can simply be translated one after another by emitting the equivalent LLVM code.

Register Structure In order to be able to reproduce the effects the instructions have on registers, a special struct is defined which contains a complete state of all registers supported by McSema on the respective architecture, each in its own variable. The struct is called `%RegState` in LLVM and `RegState` in C or C++ code. This is how the struct for x86 is defined in LLVM:

```
1 %RegState = type { i32, i32, i32, i32, i32, i32,
      i32, i32, i32, i32, i32, i32, i32, i32,
       i32, i32, i8, i8, i8, i8, i8, i8, i8,
      x86_fp80, x86_fp80, x86_fp80, x86_fp80,
      x86_fp80, x86_fp80, x86_fp80, x86_fp80, i8,
```

```
    i8, i8, i8, i8, i8, i8, i8, i8, i8, i8, i8,
    i8, i8, i8, i8, i8, i8, i8, i8, i8, i8, [10
    x i8], i128, i128, i128, i128, i128, i128,
    i128, i128, i128, i128, i128, i128, i128,
    i128, i128, i128 }
```

In C/C++, it looks as shown in the code below. This snippet has been simplified to make it more readable.

```
typedef uint32_t reg_t;

struct alignas(16) RegState
{
  reg_t RIP;
  reg_t RAX;
  reg_t RBX;
  reg_t RCX;
  reg_t RDX;
  reg_t RSI;
  reg_t RDI;
  reg_t RSP;
  reg_t RBP;

  // ... more registers following ...

};
```

Both definitions have the exact same data layout, so the memory for a single struct can be directly accessed in both languages.

Data Sections McSema simply translates data sections as global instances of packed LLVM structs. They can then be referenced from code.

Internal Functions Functions from the binary are represented as normal LLVM functions. Since functions essentially consist of a series of instructions altering registers and memory, that is exactly how McSema models them in LLVM. Each internal function has no return value, but takes a pointer to a register struct as its only argument in order to modify its contents the same way the original assembly would:

```
define internal void @example(%RegState*) #1
```

Due to this approach, the original parameters and the return value are not directly recreated in LLVM, but at the same time, the translation does not need to care for calling conventions at all in this case and can thus guarantee parameters and return values still work the same way as in the original code.

This is an example of a short function in assembly with the corresponding lifted function from which a lot of code has been omitted that would be irrellevant here:

```
start:
    add eax, 1
    ret
```

```
1  define void @example(%RegState* nocapture) #0 {
2  {
3    %RAX_write = getelementptr inbounds %RegState,
         %RegState* %0, i32 0, i32 1, !
         mcsema_real_eip !0
4
5    %1 = load i32, i32* %RAX_write, align 4, !
         mcsema_real_eip !0
6
7    %2 = add i32 %1, 1, !mcsema_real_eip !0
8
9    store volatile i32 %2, i32* %RAX_write, align
         4, !mcsema_real_eip !0
10
11   ret void, !mcsema_real_eip !1
12 }
```

At the beginning of the function, pointers to all registers are extracted from the struct to local variables, for example `%RAX_write`. To work on the value of the register, its value gets loaded into a variable called `%1`. In this example, the function adds 1 to `eax`. This calculation is performed in line 7. Finally, the calculated value is written back to the pointer in `%RAX_write` and the function returns.

In addition, McSema attaches a metadata value called `mcsema_real_eip` to most LLVM instructions which contains an address of the original assembly instruction.

Transitioning between Native and Lifted Code In order to be able to call lifted functions from other code, McSema inserts small inline assembly snippets at the beginning of the LLVM code containing a label with the name of the function to be exposed, so when the function is called from outside, this assembly code gets executed. From there, a function called `__mcsema_attach_call` (or similar, depending on the architecture) is called, which copies all needed native registers' contents to the register struct:

```
module asm "  .globl sub_8000001;"
module asm "  .globl start;"
module asm "  .type start,@function"
module asm "start:"
module asm "  .cfi_startproc;"
module asm "  pushl $sub_8000001;"
module asm "  jmp __mcsema_attach_call_cdecl;"
module asm "0:"
module asm "  .size start,0b-start;"
module asm "  .cfi_endproc;"
```

In the same way, `__mcsema_dettach_ret` exists in order to transition back into native code.

Similarly, external calls from LLVM are handled using `__mcsema_detach_call_cdecl` and `__mcsema_detach_call_cdecl` when calling a function using the cdecl calling convenction, for example. Corresponding to the definitions of the external functions in the CFG file, declarations for these are

created within the LLVM code, prefixed with an underscore:

```
declare x86_64_sysvcc i64 @_puts(i64) #2
```

These functions are implemented in inline assembly as well to perform the transition:

```
module asm "  .globl printf;"
module asm "  .globl _printf;"
module asm "  .type _printf,@function"
module asm "_printf:"
module asm "  .cfi_startproc;"
module asm "  pushl $printf;"
module asm "  jmp __mcsema_detach_call_cdecl;"
module asm "o:"
module asm "  .size _printf,ob-_printf;"
module asm "  .cfi_endproc;"
```

4.2 Exposing McSema to Python

Since McSema, which is written mostly in C++, is normally used through its command line tools, we decided to create Python bindings for it in order to be able to create Python scripts using it that are easier to understand and write than just shell scripts. In addition, this can be seen as a first step towards a complete binary analysis framework using Python, similar to Angr, but using LLVM instead of VEX.

Many different possibilities exist to expose C++ code to Python. In this case, we decided to use Boost.Python which is relatively easy to use and since McSema already uses Boost, no additional dependencies have to be added.

In our wrapper, everything exposed from C++ can be accessed through the Python module `mcsema`. It contains the class `Lifter` for translation to LLVM. In addition, `IDACFGGenerator` from `cfg_ida` can be used to recover the CFG similarly to `mcsema-disass`.

4.3 Using McSema

This is a small example which uses our Python bindings to translate an object file to LLVM:

```
from mcsema import mcsema, cfg_ida

input_file = "demo_test.o"
ida_exec = "idaq"

# Recover CFG

cfg_gen = cfg_ida.IDACFGGenerator(ida_exec, "
    get_cfg.py", input_file, "x86", "linux")
cfg_gen.entry_symbols = ["start"]
cfg_gen.func_maps = []
cfg_gen.execute("demo_test.cfg")


# Translate to LLVM

cfg_to_llvm = mcsema.Lifter("linux", "x86", "
    demo_test.cfg")
cfg_to_llvm.entry_points = ["start"]
cfg_to_llvm.execute("demo_test.bc")
```

After initializing McSema, conforming to its two step approach, we first recover the CFG using `IDACFGGenerator`. We specify which symbols to use as an entry point and then execute it. Optionally, if the binary contains any external calls, text files containing information specifying the arguments and calling conventions of the external functions have to be passed to `func_maps`. In the second step, an instance of `Lifter` is created and then executed after specifying the entry points to expose to native code.

5 Evaluation

Now it is time to use McSema on some actual code. All demos covered here are available on the accompanying git repository under `python/test`.

5.1 Demo 1 - Assembly

The first demo translates the very simple function written in assembly that has already been used as an example in 4.1.3. It simply adds 1 to a number given in `eax`:

```
add_one:
  add eax, 1
  ret
```

This code is assembled into an object file and lifted into LLVM. The lifted function, which McSema calls `sub_8000001` in LLVM, should then be called from C code. Here, we can not call it directly through the generated inline assembly, because it does not conform to the cdecl calling convention that would be used on x86. Instead, we write a custom driver in C, along with a main function to test it:

```
#define add_one_raw sub_8000001
extern void add_one_raw(RegState *);

int add_one_driver(int v)
{
  RegState        reg_state;
  unsigned long   stack[4096*10];

  memset(&reg_state, 0, sizeof(reg_state));

  reg_state.ESP =
    (unsigned long)&stack[4096*9];
  reg_state.EAX = v;

  add_one_raw(&reg_state);

  return reg_state.EAX;
```

```
}

int main(int argc, char *argv[])
{
  int a = add_one_driver(12);
  printf("%d -> %d\n", 12, a);
  return 0;
}
```

The driver function `add_one_driver` creates an instance of `RegState` and allocates some memory for the stack. It assigns the input value to `EAX` and eventually calls the raw function. In the end, it returns the calculated value from `EAX`.

This C code, in addition with a main function to test it, is being compiled with the lifted LLVM bitcode to an executable as follows:

```
clang -m32 ../../../generated/ELF_32_linux.S
    demo_test_opt.bc demo_driver.c -o
    demo_driver
```

The file `ELF_32_linux.S` contains the functions such as `__mcsema_attach_call_cdecl`. Although we do not actually use them in this demo, they are still necessary since the compilation would fail otherwise.

Running the program gives us the following output:

```
12 -> 13
```

5.2 Demo 2 - C Function

Next, we want to lift a function implemented in C:

```
int deadwing(int a)
{
  return a + 5;
}
```

This code is compiled into an object file and then lifted from that. In order to call it, it is not necessary to write any driver. Instead, the function can just be called directly. Passing the parameter into the lifted code and the return value back is all handled by McSema's functions:

```
extern int deadwing(int a);

int main(int argc, char *argv[])
{
  printf("Result: %d\n", deadwing(37));
  return 0;
}
```

Compiling and running this code results in the following output:

```
Result: 42
```

5.3 Demo 3 - Multiple Functions and External Calls

In this demo, a complete executable is being lifted and recompiled. This is the original code:

```
#include <stdio.h>
#include <stdlib.h>

int blackest_eyes(int a)
{
  return a * 7;
}

int main(int argc, char *argv[])
{
  if(argc != 2)
  {
    printf("Usage: %s [number]\n", argv[0]);
    return 1;
  }

  int a = atoi(argv[1]);
  a = blackest_eyes(a);

  printf("The result is %d.\n", a);
  return 0;
}
```

In this case, the CFG Generator needs additional knowledge about `printf` and `atoi`. McSema already provides a file containing information about the most common functions on Linux, located at `tools/mcsema_disass/defs/linux.txt` along with a similar file for Windows. This has to be passed to the CFG Generator class in the Python script:

```
# ...

cfg_gen = common.cfg_generator("demo_test", "x86"
    , "linux")
cfg_gen.func_maps = ["../../../tools/
    mcsema_disass/defs/linux.txt"]
cfg_gen.entry_symbols = ["main"]
cfg_gen.execute("demo_test.cfg")

# ...
```

After lifting the executable using `main` as the entry point, the bitcode can be directly compiled again:

```
clang -m32 ../../../generated/ELF_32_linux.S
    demo_test_opt.bc -o demo_recompiled
```

Running this recompiled program shows the same behaviour as the original one:

```
$ ./demo_recompiled 191
The result is 1337.
```

5.4 Demo 4 - Jump Table

This example demonstrates how McSema handles jump tables. As a basis serves a function containing a switch statement with evenly distributed cases:

```c
int switch_func(int a)
{
  switch(a)
  {
    case 0:
      printf("42\n");
      break;
    case 1:
      printf("1337\n");
      break;

      // ... cases 2 to 6 ...

    case 7:
      printf("4563");
      break;
    default:
      break;
  }

  printf("end of switch_func\n");
  return 12345;
}
```

Compiling this function using a common compiler will most likely result in a jump table being created. This is an excerpt of the assembly code clang would generate:

```
mov ecx, dword [eax*4]
jmp ecx

; ecx == 0 => jmp above would end up jumping here
lea eax, [0] ; RELOC 32 .L.str
mov dword [esp], eax
call reloc.printf_115
mov dword [ebp - 0x10], eax
jmp loc.08000116

; other cases following here ...
```

Given the value of a in ecx, instead of performing a number of comparisons, the raw value is multiplied by 4 and directly used as a jump offset. Given that it is correctly represented in the CFG file, McSema can handle this type of code. After lifing the function, it can be observed that a number of callback functions are created, each calling an internal function containing the code for one case. In addition, a global struct is created containing pointers to these functions:

```
%0 = type <{ void ()*, void ()*, void ()*, void ()*, void ()*, void ()*, void ()*, void ()* }>

@data_0x80000fc = internal constant %0 <{
  void ()* @callback_sub_8000029,
  void ()* @callback_sub_800003f,
  void ()* @callback_sub_8000055,
  void ()* @callback_sub_800006b,
  void ()* @callback_sub_8000081,
  void ()* @callback_sub_8000097,
  void ()* @callback_sub_80000ad,
  void ()* @callback_sub_80000c3 }>, align 32
```

Calling the correct function for the input value is implemented in the following snippet:

```
%43 = shl i32 %42, 2

%44 = add i32 %43, ptrtoint (%0* @data_0x80000fc to i32), !mcsema_real_eip !11

%45 = inttoptr i32 %44 to i32*, !mcsema_real_eip !11

%46 = load i32* %45, align 4, !mcsema_real_eip !11

store i32 %46, i32* %XIP, align 4, !mcsema_real_eip !12

tail call void @__mcsema_detach_call_value(), !mcsema_real_eip !12
```

Given the value in %42, it is first multipled by 4 using a shift operation to get the right offset in the 32-bit aligned struct. Then, the base address of `@data_0x80000fc` is converted to an int, added to the offset and converted back to a pointer. The pointer to the callback function is loaded from this address and stored into the instruction pointer inside the register struct. The function `__mcsema_detach_call_value()` will then take this address from the struct and eventually call the function.

Unfortunately, at the current state, recompiling the code and calling the function results in a segmentation fault after the code from the case has been executed. This appears to be due to a bug in McSema as `demo13` from the official McSema demos shows the same problem.

5.5 Demo 5 - Solving a CrackMe with KLEE

The last demo is a very interesting one. We have a small program in the form of ab 64-bit ELF executable that asks for a password encoded in base64:

```
$ ./qual
Welcome. Please enter your base64 encoded input:
pleasechangeme
Nope.
```

Our goal now is to lift the whole program to LLVM and then use KLEE on it to find out the password. As a first step, we take a look inside the assembly code to get an overview of the program's structure, which can be done using radare2, for example. [7]

There are only two functions inside the binary that are interesting in our case: `main` and `b64d` which decodes a base64 encoded string and would originally have a signature like this:

```c
int b64d(char *b64_string, void **out_data);
```

It takes the string in `b64_string`, allocates the memory for the decoded data on the heap and

writes the pointer to it into `*out_data`. The return value is the number of bytes decoded.

The `main` function first reads a string through `fgets` and then calls `b64d` to decode it. It checks if the length is exactly 16 and otherwise fails immediately. After that, it performs a number of xor and circular bit shift operations on the data inside a loop and compares the outcome to some predefined values. Solving the equation system that these computations yield by hand would be practically impossible as well as trying out all 2^{128} possibilities for the 16 bytes. That is why we will apply symbolic execution through KLEE here.

Knowing all this, we can now lift the binary to LLVM. This is simply done using `main` as the entry point and supplying McSema's default external function definitions for Linux to the CFG generator. Recompiling the original bitcode shows that our LLVM code still behaves like the original binary, at least for wrong passwords.

Looking at the LLVM code, we see that there is the `main` function, called `sub_40079f`, and the internal function `b64d`:

```
define x86_64_sysvcc
  void @sub_40079f(%RegState*) #0 {
  ; ...
}

define x86_64_sysvcc void @b64d(%RegState*) #0 {
  ; ...
}
```

In addition to that, there are the inline assembly snippets generated by McSema at the top to transition between native and lifted code for all external functions and `main`:

```
module asm "  .globl sub_40079f;"
module asm "  .globl main;"
module asm "  .type main,@function"
module asm "main:"
module asm "  .cfi_startproc;"
module asm "  pushq %rax;"
module asm "  leaq sub_40079f(%rip), %rax;"
module asm "  xchgq (%rsp), %rax;"
module asm "  jmp __mcsema_attach_call;"
module asm "0:"
module asm "  .size main,0b-main;"
module asm "  .cfi_endproc;"
```

This is a problem here, since we would like to run the code through KLEE, which can only operate on actual LLVM code. Thus, it is necessary to implement driver functions like in Demo 1. First, we remove all the inline assembly. Then, we write a simple driver for `main`, setting up the stack and register struct, passing `argc` and `argv` and returning the original return value from `RAX`:

```
unsigned long stack[4096*10];
RegState reg_state;

#define qual_main_raw sub_40079f
extern void qual_main_raw(RegState *);

int qual_main_driver(int argc, char **argv)
{
  memset(&stack, 0, sizeof(stack));
  memset(&reg_state, 0, sizeof(reg_state));

  reg_state.RBP = 0;
  reg_state.RSP = (unsigned long)&stack[4096*9];

  reg_state.RDI = (unsigned long)argc;
  reg_state.RSI = (unsigned long)argv;

  qual_main_raw(&reg_state);

  return (int)reg_state.RAX;
}


int main(int argc, char **argv)
{
  return qual_main_driver(argc, argv);
}
```

For all external functions, we can simply call the original functions. One thing that is necessary here though is to add 8 to the stack pointer in the register struct, because the lifted code always pushes a return address to its stack which would not be cleared otherwise:

```
int _puts(const char *s)
{
  reg_state.RSP += 8;
  return puts(s);
}
```

For convenience, fgets should automatically return a stub string, so the program does not stop to wait for input every time:

```
char password[] = "thisistheinputstring....";

char *_fgets(char *s, int size, FILE *stream)
{
  reg_state.RSP += 8;
  printf("Skipping fgets with stub data.\n");
  strcpy(s, password);
  return s;
}
```

Compiling the program in this state shows that it still seems to work:

```
Welcome. Please enter your base64 encoded input:
Skipping fgets with stub data.
Nope.
```

So far, so good. Now that all inline assembly code has been replaced, it is possible to run the program in LLVM through KLEE. However, it is still necessary to tell KLEE which memory to handle as a symbolic variable. To do so, we will replace the internal function `b64d`, so it will always return a 16 byte long memory chunk marked as symbolic.

For this, we implement the function `b64d_fake` in C, which operates on a register struct:

```
void b64d_fake(RegState *reg_state)
{
  reg_state->RSP += 8;
  char *b64_str = (char *)reg_state->RDI;
  unsigned char **out_buf =
      (unsigned char **)reg_state->RSI;

  unsigned char *buf =
      (unsigned char *)malloc(16);
  klee_make_symbolic(buf, 16, "password");
  (*out_buf) = buf;

  int ret = 16;
  reg_state->RAX = (unsigned long)ret;
}
```

To make sure our function gets called instead of the original one, we declare it inside the lifted LLVM module and replace the call to `b64d` with `b64d_fake`:

```
declare x86_64_sysvcc
    void @b64d_fake(%RegState*) #0

define x86_64_sysvcc
    void @sub_40079f(%RegState*) #0
{
  ; ...
  tail call x86_64_sysvcc void @b64d_fake(
      %RegState* %0), !mcsema_real_eip !18
  ; ...
}
```

After compiling our C code with the LLVM code to a single LLVM bitcode file, we are ready to execute KLEE:

```
clang [...] -emit-llvm -c driver.c \
        -o driver_klee.bc
llvm-link driver_klee.bc qual_opt.bc \
        -o driver_klee_linked.bc
```

Running KLEE on the bitcode shows it generated four test cases:

```
$ klee driver_klee_linked.bc
...
KLEE: done: total instructions = 1072762
KLEE: done: completed paths = 4
KLEE: done: generated tests = 4
```

These test cases should cover all possible control flow depending on the symbolic variable. So if a correct password exists, one of the test cases should contain it. Using `ktest-tool`, it is possible to read out example values for each test case. For example, test case 4 gives us the following information:

```
$ ktest-tool klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['driver_klee_linked.bc']
num objects: 1
object    0: name: 'password'
object    0: size: 16
object    0: data: '\xb2\xd0cu\x90\x14\xcfB\xf5\
    xd9o\xf1\x10\xdb-\xa7'
```

The escaped string shown as data encoded in base64 is `stBjdZAUz0L12TDxENstpw==`. We can now run the original binary and see if we have found the right solution:

```
$ ./qual
Welcome. Please enter your base64 encoded input:
stBjdZAUz0L12TDxENstpw==
Congratz, you win!
Please send your solution to kirschju@sec.in.tum.
    de
```

6 Conclusion

We have shown how McSema translates machine code to LLVM and used it in a number of practical demos including performing symbolic execution on lifted code. What can be seen from these examples is that, once a correct representation of the code in LLVM has been obtained, it can be used and modified in many ways very flexibly. It is also easily possible to recompile this code to the original architecture and, with some additional effort, it would even be possible to compile it for any other architecture or platform as long as all used external functions are present.

McSema's lifted code works on virtual registers, in the form of variables inside the register struct. When comparing McSema's LLVM code to VEX, it can be observed that what McSema simulates on top of LLVM is actually very similar to VEX's built-in functionality of simulating register state. However, while VEX is bound very closely to the machine code, LLVM IR provides much more high level concepts such as defined functions with parameters. In addition to that, LLVM IR is designed from the beginning to be highly transformable while VEX aims to always transparently represent the original code.

6.1 Limitations

Because the machine code instructions directly translated to LLVM result in not one single, but usually a whole series of LLVM instructions and a fair amount of boilerplate code is present, the lifted code gets much harder to read and is thus less suited for analysing the code by hand. While the LLVM optimizer in its current form helps eliminating code which is definitely not needed, a certain amount of noisy code still remains.

Also, because of McSema's naive translation approach, variables and function parameters which were originally compiled from C code for example

are not represented as LLVM variables and parameters, but indirectly through memory contents and the register struct.

At the current state, McSema does not support self-modifying code and exceptions. It is also limited to x86 and x86_64 and while it supports the most important instructions, including FPU and SSE, there are still many missing.

6.2 Future Outlook and Other Projects

For the future, a potential way to counteract the remaining problems could be implemented by creating new LLVM passes for these kinds of tasks. Doing so could be a modular way to make the code more high-level again step by step. The ultimate goal would be to have a fully functional decompiler to LLVM that generates easy to read code.

To extend McSema's functionality, it is possible to independently implement the semantics translation for single instructions because of the way McSema's code is built. In addition to that, Trail of Bits have announced to implement ARM support in 2017 as part of a blog post [8].

Another project which has to be mentioned is Remill, developed by the same company as McSema, Trail of Bits. At first glance, both projects look very similar. Remill seperates CFG recovery from translation just like McSema, for example. But Remill's representation of assembly code in LLVM is very different. While McSema lifts assembly functions as LLVM functions, Remill creates a seperate LLVM function for each basic block in the CFG. In general, while McSema aims to provide a representation of the program as high level as possible, Remill does translate to LLVM, but tries to stay very close to the original assembly code. This makes McSema better suited for tasks like extracting functionality from binary code to integrate in own code or partially decompiling and transforming code, while Remill might be a better choice for dynamic program analysis tasks such as symbolic execution. So, in conclusion, if one would like to build a framework similar to Angr, but with LLVM instead of VEX, Remill seems to be the primary choice. [3]

6.3 Acknowledgements

Finally, I would like to thank Peter Goodman, Ryan Stortz and everyone else on the Empire Hacking Slack who answered my questions and and helped me working on this thesis.

References

[1] Llvm language reference manual. `http://llvm.org/docs/LangRef.html`. [accessed 28-January-2017].

[2] Mcsema git repository for this thesis. `https://github.com/thestr4ng3r/mcsema`. [tag thesis2].

[3] Remill. `https://github.com/trailofbits/remill`. [accessed 28-January-2017].

[4] Remill documentation. `https://github.com/trailofbits/remill/tree/9a5f684dd3acadf1eb534646e58745c303f58213/docs`. [commit 9a5f684, accessed 28-January-2017].

[5] Valgrind source code. `http://valgrind.org/downloads/current.html`. [version 3.12.0, accessed 28-January-2017].

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[7] Artem Dinaburg. Close encounters with symbolic execution. `https://blog.trailofbits.com/2014/11/25/close-encounters-with-symbolic-execution/`, 2014. [accessed 28-January-2017].

[8] Dan Guido. 2016 year in review. `https://blog.trailofbits.com/2017/01/09/2016-year-in-review/`, 2017. [accessed 28-January-2017].

[9] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.