DECOMPILING BINARIES INTO LLVM IR USING MCSEMA AND DYNINST

LUKÁŠ KORENČIK

DIPLOMA THESIS

2019

Faculty of Informatics
Masaryk University

SUPERVISOR
RNDr. Petr Ročkai, Ph.D.

DECLARATION

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference to its source.

ABSTRACT

There are many tools that operate on LLVM bitcode. To use these tools, LLVM bitcode or original source code are required (LLVM bitcode can be obtained from the source code by a compiler). Sometimes however, only already compiled binaries are available – there is no standard and well-defined process to obtain LLVM bitcode from a binary.

McSema is a tool that translates binaries into LLVM bitcode; it makes the tools applicable on previously unavailable targets. McSema itself is open-source, although it relies on proprietary third-party libraries to provide disassembly capabilities. This is problematic, as it prevents many users from using the software.

This thesis provides alternative implementation to the proprietary component of McSema which uses open-source Dyninst disassembler. With this new implementation McSema can be used without proprietary software. The performance of the open source version is demonstrated on a set of programs and the results are compared with already existing components.

KEYWORDS

McSema, Dyninst, LLVM, Disassembly, Decompilation, Assembler, ELF, Processor architecture, Binary analysis

ACKNOWLEDGEMENTS

First, I would like to thank you Mornfall, for all your guidance and patience.

Second, I would like to thank everybody with access to the ParaDise laboratory for creating unique working atmosphere; especially the DIVINE corner.

Third, I would like to thank Peter and guys over at the ToB for their answers to my questions during development – there were quite many of them.

Finally I would like to thank my family and friends for supporting me and having patience with me.

CONTENTS

1 INTRODUCTION

The result of compilation is typically an executable binary. After the compilation is done, the source code is no longer needed to run a program – it often happens that only binaries are being distributed. Many tools require the original source code as an input; if only the binary is available these tools cannot be used. McSema [Din+19] can be used to help in this case. It translates binaries to LLVM bitcode (intermediate representation of a compiler) which makes the previously unavailable tools available.

While McSema itself is open-source, one of its components (frontend) is dependent on proprietary tools. The frontend retrieves information from binary needed for successful translation into LLVM bitcode, such as functions and their instructions or sections and data in them. This information is then encoded in McSema specific message format. The proprietary tools are used for actual information retrieval from the binary; the frontend aggregates and checks the information.

Currently supported frontends are based on IDA Pro [Hex19] and Binary Ninja [Vec19] which are not only proprietary but also require commercial license. This is problematic both for users and developers of other open-source software; incorporation of the tool depending on proprietary software is not really desired.

This thesis solves the issue. – Implementation of the frontend which uses open-source diassembler Dyninst [Par19] is presented.

Compared to the frontend based on IDA Pro, its scope is smaller – only 64-bit ELFs and programs that were originally implemented in the C programming language are supported. The goal is for the Dyninst frontend to be as good as the IDA Pro frontend on this set.

Chapter 2 introduces the reader to LLVM bitcode [LLV19]; its syntax and semantics are explained. Assembly languages and machine code are described in Chapter 3. Chapter 4 describes the traditional compilation process (`clang` compiler for example) and differences in abstraction levels between languages. McSema is described in Chapter 5: architecture, specification of the frontends and the lift process itself. Chapter 6 describes implementation of the Dyninst frontend and evaluation of all three frontends on a set of programs. Finally, Chapter 7 summarizes the contribution of the thesis.

2

LLVM

From the LLVM language reference manual [LLV19]:

> LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing "all" high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.

The LLVM Project [Lat19] is a collection of modular and reusable compiler and toolchain technologies. It consists of a set of libraries for building and manipulation of an intermediate representation (IR) of a program and several platform-specific code generators.

LLVM IR can be represented in three different forms. First two are on-disk representations, compact serialized bitcode (`.bc` files) and human readable representation (`.ll` files). LLVM project provides tools to convert between these representations effortlessly. Last representation is in memory as C++ objects that can be manipulated using API provided by LLVM libraries. This representation can be easily read from or serialized into the other two.

2.1 BITCODE INTRODUCTION

Human readable bitcode representation is similar to assembly language, providing some extra verbosity and types. A single LLVM IR file, called a module, corresponds to one translation unit. It contains description of global data, function definitions and metadata. Multiple modules can be linked together using the LLVM linker.

There are two basic types of identifiers. `@` for global identifiers (function names, global variables) and `%` for local identifiers (register names, types). These identifiers can either be named or unnamed, represented by an unsigned numeric value. A special case are constants, representing constant values using different syntax based on the data type of a constant.

2.1.1 *Functions*

An LLVM function definition consists of a header and a body. The header provides information about name, type and number of parameters, attributes. Function attributes are used to communicate additional information about a function, that does not affect the semantics of the function. The body is formed by basic blocks which form a control flow graph.

A basic block may start with a label, which serves as its name. Labels can be used to reference the block, for example in branching instructions or φ nodes. The main body of a basic block is a contiguous sequence of instructions without any branching. The instructions are executed sequentially in order of their appearance inside the block [Roč15]. The last instruction in a well-formed basic block must be a terminator which is for example an instruction that explicitly transfers control flow to a different basic block (a branching instruction) or exits the function (`return` instruction) [LLV19]. The instructions inside each basic block operate on values in virtual registers, or they can move values between registers and memory.

Each instruction has at most one return value, which is always assigned to a new virtual register for the duration of execution of a function. The instruction can take any number of inputs, all of which must reside either in virtual registers or be in the form of constant. There can be an unlimited number of registers used inside the function.

2.1.2 *SSA*

> Static single assignment is a property of an intermediate representation, which requires that each variable is assigned exactly once, and every variable is defined before it is used [Cyt+91].

Virtual registers are in SSA form, as they are assigned (defined) only once. From the entire module only virtual registers are in SSA form, which means the module itself is in partial SSA form.

Since virtual registers are in SSA form their address cannot be taken. This makes them a suboptimal representation of local variables. To solve this problem, LLVM uses the `alloca` instruction. `alloca` creates separate space to store address-taken variables and returns a pointer to it (usually allocated on the stack when compiled). It is possible to operate on this pointer as well as the value inside the allocated memory itself (`load`, `store` instructions). These values are not in SSA form.

SSA form also requires φ instructions (nodes). These instructions are used in the beginning of the basic block to merge values from different

basic blocks that are predecessors of the current one (they have the current block as a target in their terminator). The semantics of a φ node is to create a new register with a value dependent on which of the predecessors transferred control to this block.

```
%30 = phi i64 [ 42, %entry_bb ], [ %17, %another_bb ]
```

In this example register %30 has either value 42 or copy of value in %17 depending on predecessor.

2.1.3 *Types*

LLVM is statically typed language: both values and variables are typed. There are primitive types, for example integer types of arbitrary width (`i64` is 64 bit width integer type, `i1` is used for booleans), floating-point numbers (`float`, `double`) or pointers (`i64*`, a pointer to a 64 bit integer).

To support aggregate types, LLVM also has more complex types. Array types, similar to C-like arrays, which represents a contiguous block of memory, for example `[i32 x 10]` corresponds to array of ten integers.

Structure types correspond to structures and classes in higher languages. They can be composed from multiple elements of any other type. A type which represents a structure containing two 32-bit integers and a pointer to another structure:

```
%custom_type = { i32, i32, %another_struct_type* }
```

Access to member fields of structure is differs for values stored in memory and in virtual registers. Elements of structures in registers are accessed using `insertvalue` and `extractvalue` instructions which takes at least one integer as an argument to calculate offset. The integer specifies the offset of the field member in the structure (`0` is the first member field, `1` the second, etc.).

Following example retrieves second member of a structure that is in register %12:

```
%second = extractvalue { i32, i32 } %12, 1
```

Symmetrically to retrieve an element of structure in memory, `gep` (`getelementptr`) combined with `load`, `store` instructions is used [LLV19].

LLVM provides a variety of casting instruction to allow casting between types. Pointer type can be casted to a different pointer type using the `bitcast` instruction. The `inttoptr` and the `ptrtoint` instructions allow conversion between pointers and integer types. Casting integer types to integer types of different size is done by `zext` (zero extend),

`sext` (signed extend) and `trunc`, depending on the size and signedness of the source and target types.

Example 2.1.1. *Consider the following function that sums numbers from zero to n in the C language:*

```
int sum(int n) {
  if (n == 0){
    return 0;
  }
  return n + sum(n - 1)
}
```

Possible implementation in LLVM (standard compiler would optimize the recursion away):

```
define i32 @sum(i32 %n) {
entry:
  %0 = icmp eq i32 %n, 0
  br i1 %0, label %then, label %else
then:
  ret i32 0
else:
  %1 = add i32 %n, -1
  %2 = call i32 @sum(i32 %1)
  %3 = add i32 %n, %2
  ret i32 %3
}
```

In LLVM, the name of the function `sum` *is prefixed with* `@`*, signaling that it is global object. Return and parameter types are both* `i32` *which corresponds to* `int`*. The body of the function starts with a labeled block* `entry`*. If the block did not have an explicit label, it would be given the name* `%0` *and first register would be named* `%1`*.*

In this case, register `%0` *holds the result of the* `icmp eq` *instruction which compares both its operands for equality. Jump is performed by the* `br` *instruction, based on the value of* `%0` *and chooses* `%then` *in the case* `%0` *is true which means that operands of* `icmp eq` *were equal, or the* `%else` *otherwise.*

The same control flow decision happens in the original C code, but is less visible in higher language. In LLVM there are also no loops, the entire control flow is represented using branching over basic blocks.

The `%then` *block simply returns* `0`*. The basic block* `%else` *computes a new decremented argument for recursive call (the* `add` *instruction with* `-1`*) and the call itself is done by the* `call` *instruction. After recursive call finishes, its return value (*`%2`*) is added to the value of the argument of current function (*`%n`*) and returned.*

3 ASSEMBLY / MACHINE CODE

Assembly is a family of low level programming languages, slightly more abstract than machine code. Compared to machine code it abstracts from addresses (symbolic labels are used instead of addresses) and instructions are encoded in a human readable format using mnemonics for opcodes instead of plain sequence of bytes. The tool to build machine code from assembly is usually called an assembler.

Languages from this family differ based on the target processor architecture or chosen assembler. This chapter tries to outline core characteristics that most of them share.

The term subroutine is used to describe any sequence of instructions that can be called and ends with return. Functions are subset of subroutines and correspond to functions in higher level programming languages. Difference is for example that subroutines do not have to return control flow to the call site of the caller, but to any address.

3.1 REGISTERS

Physical register is a part of the CPU. It is used as small, fast storage during the execution of the processor. There are usually only a few register names available to a programmer, however the processor may contain a larger number of physical registers. Their size and number is architecture dependent. To execute an instruction, its operands must be stored in the registers [Int].

A register in an assemly program does not have to be the same physical register at every point of the execution of the program. The rest of the thesis uses the term physical register for the CPU component and the term register for the name available to a programmer and used in the assembly.

Several values that are very often needed during an execution of subroutines or are important for the processor, are stored in specially reserved registers. Classical example would be the instruction pointer (IP) which represents the address of the next instruction to be executed. The address is accessed at least once per instruction, as a result of the need to decode the next instruction. The stack pointer (SP) is not used as often as the IP, but it is needed in almost every subroutine, since every subroutine implementing non-trivial functionality needs to access the stack.

It is common to use separate registers for flags, such as for example results of comparisons or overflows. The floating point unit (FPU) registers form a special category of registers. They are used to store floating point values and often operands of the floating point instructions must reside inside these registers [Amdb].

Many systems contain segments registers which used to have special use with the segmented memory model. Modern systems however use flat memory model, which frees these register to other usage.

3.2 MEMORY

Assembly program uses virtual memory. This allows usage of the entire address space available to programs. Part of the memory assembly program uses is called stack. The maximum size of the stack is static when program is executed, it cannot grow indefinitely. Top of the stack is available to program via the stack pointer that is typically stored in a reserved register. Therefore if a different value is stored in the SP the current top of the stack changes.

Except for explicit arithmetic operations on the pointer, assembly languages commonly contain instructions which implement basic operations manipulating the stack.

- `push`: New value is stored at the location stored in the SP and the pointer itself is adjusted by the size of the newly stored value.
- `pop`: Value at the location pointed by the SP is effectively removed (copied to register/different memory) and the pointer itself is adjusted by the size of the removed value.

Every function typically creates a function frame on the stack, where all locally needed data are stored. In practice this means local variables, function arguments and the return address (see Section 3.5.2).

Global values are referenced by labels; therefore, they do not need memory locations. In the actual machine code they will be given addresses somewhere in the data segment of memory.

3.2.1 *OS view of memory*

Memory of a running program is split into more segments than just the stack, but the assembly program does not require any information about it. Exact partitions and directions of growth depend on the operating system. Older systems (16-bit) may use the segmented memory model instead of the flat memory model [Amda].

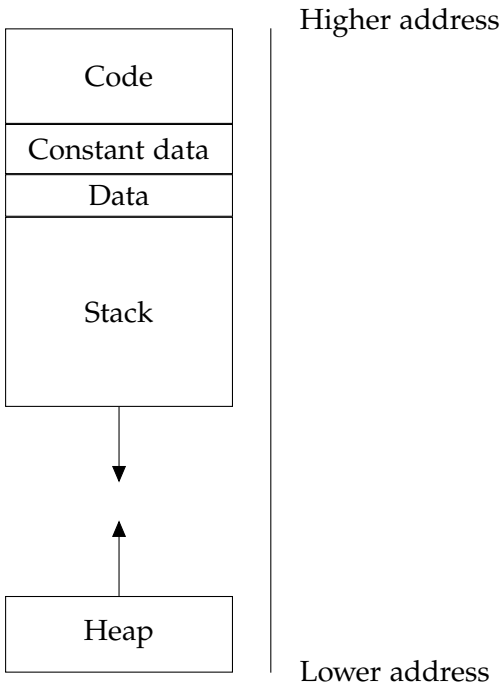

Figure 3.1: Possible memory layout of the program

Stack is a contiguous block of memory, that is used by functions to store their local variables together with other information needed to allow proper control flow.

Data typically contains two sections *.data* and *.bss*. Former section contains initialized global objects, while latter contains global objects that are zero initialized by default. Neither of these sections can be expanded during the execution of the program, their size is static.

Heap contains dynamically allocated data.

Code (Text) contains machine code of the currently running program. This segment does not have to be read-only (although it usually is) which allows programs to rewrite themselves during their execution.

Constant Data contains Objects that cannot be modified during an execution of the program. It can be included in the text segment if the text segment is read-only; otherwise, it forms its own section.

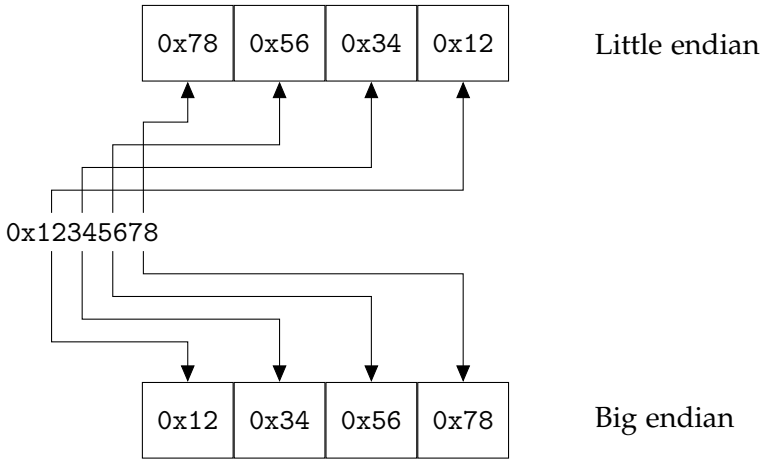

Figure 3.2: 4-byte number `0x12345678` stored at address `0xa0`

3.3 ENDIANNESS

Endiannes is the byte order in which bytes are arranged when stored in memory. Two formats are commonly used, although others also exists (middle-endian).

Big-endian stores the most significant byte on the lowest address (is stored first). Little-endian works in reverse: least significant byte on lowest address (most significant byte last). These two formats are for obvious reasons not compatible. Each architecture uses primarily one fixed format but it can provide instructions to work with other formats as well.

Figure 3.2 shows how 4-byte wide number is stored in both formats. Advantage of little-endian format is that from the same address correct values of different width can be read. Number stored in the example has 4 bytes, but in the little-endian encoding if only 2 bytes were read from `0xa0`, they would form a correctly truncated 2-byte wide number from the original value (the most significant bytes are forgotten). Several mathematical operation are more natural with numbers in little-endian as they often work first on least significant bytes moving towards the more significant bytes. Memory addresses nicely correspond to this, with steady increase in offset from beginning.

Big-endian encoding stores the most significant byte first which allows a quick test whether stored number is positive or negative, since needed bit is in this byte. It also corresponds with how humans write down numerical values which makes this representation more easily human readable.

3.4 INSTRUCTIONS

Instructions are the smallest units assembly programs are built from. Compared to the higher level languages it often takes several instructions to do a relatively simple task. Each instruction is represented as mnemonic in the given language. Mnemonics do not have to be shared between languages as each processor may support different set of instructions. This section uses the AT&T syntax (see Section 3.7.1.1) to demonstrate the examples.

Number of operands is different for every instruction and can have arbitrary arity, although usually instructions take at most two operands. Operands of an instruction can be immediate values, registers or memory addresses [Amdb]. Immediate values are used for constants, for example in the following instruction,

```
add 0x2, %rax
```

constant number `0x2` will be hardcoded in the machine code itself. Instructions mnemonic specifies the operation – `add` sums the operands and stores the result in the second.

3.4.1 *Memory addressing*

If the operand of the instruction is memory, it can have one of the following forms [Amda].

Absolute address: The memory address is constant and can be determined by the assembler. In the machine code it would be represented by an absolute value (for example `0x400650`), in the assembly label is used instead.

Addressing relative to register: Content of a specified register (base register) is used as the target address. The address can be further modified with the following options:

Immediate offset: Constant offset applied to the content of the base register.

Index register: Modify the content of the base register with multiple of the value stored in the index register (by a specified scale factor).

Typical example would be the frame pointer. To access a local variable stored on the stack, a function can use the address stored in the frame pointer register (if it is available) and modify it by known offset to get the address of the desired variable.

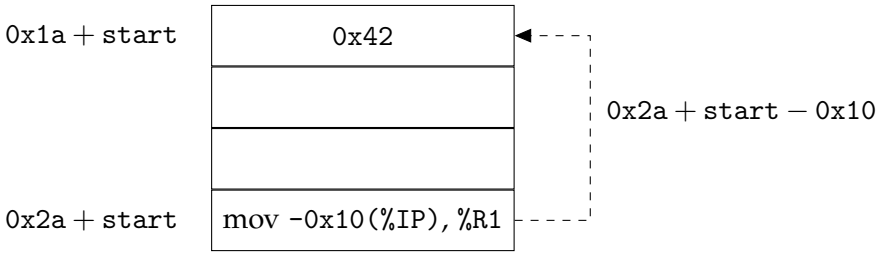

Figure 3.3: Relative addressing

IP-relative addressing is a special case of the absolute addressing in which the constant is not represented by a fixed number but as an offset to the current IP. The result is a constant, since the value of the IP is a constant at each instruction and addition of a constant offset still results in a constant value. This allows more flexible representation of the absolute addressing in the program.

3.4.2 *Position Independent Code*

When generating machine code, absolute address may be used to reference object in the data section. Every time the program is executed, the data sections must be mapped to the same addresses, so that the absolute addresses used by instructions are valid. This can become a problem once shared libraries are taken into account, since there is no guarantee that the fixed addresses of their separate sections will not collide.

Program that is position independent replaces every absolute address with `IP + offset`, where the `IP` is the address of a current instruction and the `offset` is a constant. Therefore since no absolute addresses are used and everything is relative to the address of the current instruction, it does not matter where in the memory program is loaded, offsets will always be the same [Lev99]. Figure 3.3 shows an example.

Position independent code can be used in executables as well, to provide randomized address space layout which improves the security of the program.

3.4.3 *Control flow*

Control flow can be altered by instructions from the jump family. The simplest is a direct unconditional jump which takes label of a basic block and sets current instruction pointer to the first instruction of the block.

Special case of an unconditional jump is call to a subroutine with the `call` instruction. Before jumping to the entry block of called subroutine

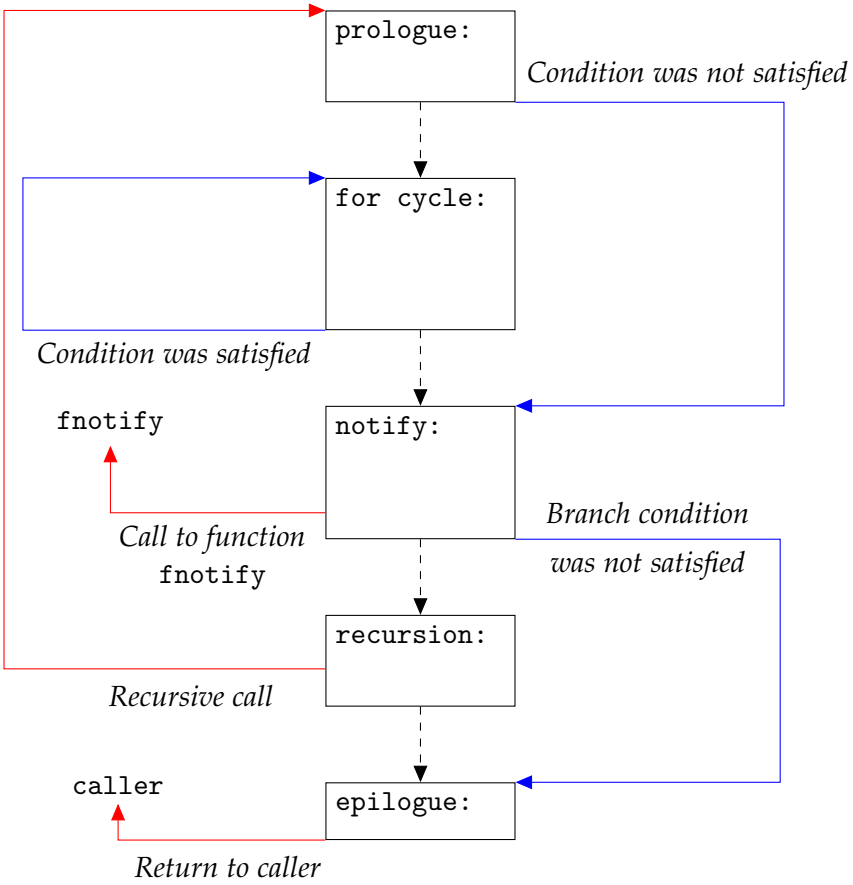

Figure 3.4: Example of control flow graph of function.

(callee), it typically pushes the address (return address) of next instruction in the current block. After the callee finishes its execution it uses the `ret` instruction to transfer control flow back to the caller. The `ret` instruction pops the stack and jumps to the popped value.

Conditional jump transfers control flow if a specific condition is met. There are typically several jump instructions for the most common cases of comparisons, for example jump if equal, jump if greater, etc. Jumps themselves are normally not able to do comparisons; therefore, they depend on instructions that are executed before them to do the comparison and store the result in the predetermined location. If the appropriate register contains value that satisfies condition of the jump, the control flow is transferred to the operand of the jump. Otherwise next instruction is executed – the first instruction of the next block, since the jump instruction terminates basic blocks.

Sometimes the target of a jump instruction cannot be statically determined, since it was passed from another subroutine or there can be more of them (jump table may be generated from the `switch` statement in the original program). Indirect jump is used in this case: rather than stating the target explicitly in the argument, the location which stores the target is used as operand. The target address is loaded from the location (memory, register) and control flow is transferred there. Indirect call works analogously.

3.5 SUBROUTINE

Similar to all other languages basic logical unit of a program is a subroutine – it can, but does not have to correspond to functions in higher level languages. It is made of basic blocks. Basic block is a sequence of instructions with property, that if the first instruction in the block is executed the rest must be executed exactly once in order of their appearance. Only the first instruction can be the target of a jump instruction anywhere in the program. Basic block does not have to be ended by an instruction that alters the control flow. In the case such instruction is not present, program will continue with the next block by their appearance in the code.

Basic blocks which form one subroutine do not have to be stored at contiguous addresses, neither holds that every basic block must belong to exactly one subroutine. Figure 3.5 demonstrates this. Two subroutines have different code, but their last block implements the same functionality, therefore there it is enough to generate just one block shared by them.

3.5.1 *Calling Convention*

Calling convention for chosen pair of architecture and operating system standardizes how calls of functions are implemented. This standard among other things specifies how arguments are passed (order of registers, which types are passed via the stack), whether the caller or the callee is responsible for the stack cleanup and how the return value is returned. Unfortunately these standards can be very different from one operating system to another [Jan14].

Following the calling conventions makes compatibility easier. Instead of always compiling all sources, the libraries can be compiled only once (different compiler can be used as well) and then simply linked to a program which uses them. Since functions inside the library are following the convention, a program which calls them properly will work.

```
void f(i32 i, i32 j, i64* callback) {
  if (i == 1) {   // f0
   ++i;       // f1
  }
  callback(i, j)  // gf0
}

void g(i32 i, i32 j, i64* callback) {
  if (j == 1) {    // g0
    --i          // g1
  }
  callback(i, j)  // gf0
}
```

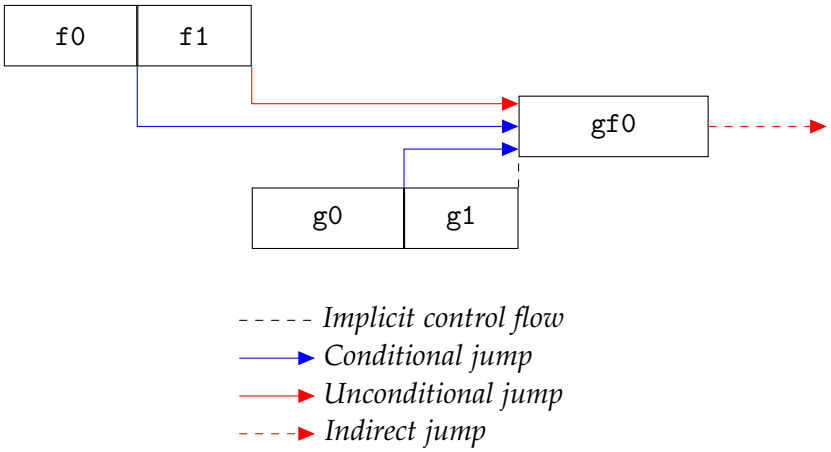

Figure 3.5: Example of functions sharing basic block.

Since all functions share the same set of registers, it will happen that caller and callee will both use the same set. To tackle this problem several registers are preserved accross function calls, i.e. values stored inside them after the `call` instruction must be the same as before. The callee often needs to use some of the preserved registers; therefore, it needs to store the original value first, use the register for its own calculation and restore it right before returning. If the caller needs to preserve value of some register that is not marked as preserved in the calling convention, the caller needs to store and restore it after the call itself. Content of registers is typically pushed to the stack and poped later to restore the original value.

3.5.2 *Function prologue & epilogue*

Every function needs to setup its own stack frame before it can use the stack. As a result the function typically starts (differs by architecture) with the same sequence of instructions called a *function prologue*.

- Push all preserved registers to the stack.
- Push the frame pointer in case it is not preserved across calls.
- Move the current SP into the frame pointer.
- Modify SP by N which is the size of the frame.

Function epilogue is responsible for returning the control to the calling function. It typically reverses the prologue.

- Pops all preserved registers.
- In case the SP was not preserved, move the frame pointer into the SP.
- Return instruction (pop return address from the stack and jump on it).

Function prologue and epilogue can also contain code to improve protection of the stack against attacks, such as for example the buffer overflow attack [TF01].

3.6 ARCHITECTURE DEPENDENCY

Unfortunately an assembly language is not architecture agnostic, since language (code) written for one processor may not run on a different one. There are several reasons for this incompatibility.

Register set available depends on the target architecture. The 64-bit architecture amd64 (also known as x86_64) will contain bigger set of registers than 32-bit x86 and totally different set than aarch64.

However even if the target architectures are the same, it does not guarantee compatibility between all models. For example Advanced Vector Extensions adds several new `YMM` (256-bit width) registers that are not present in earlier versions of Intel processors. Therefore assembly code written for one of the newer processors will not run on an older model which does not contain the `YMM` registers. Also instructions which older models do not support are added with newer revisions.

3.7 x86, x86_64

x86 is a processor architecture. It is based on the Intel 8086 microprocessor. The 64-bit version is called x86_64 (amd64) and is an extension of the 32-bit version with several improvements [Amda] (original is the 16-bit version). Most modern desktop computers and servers use processors with some version of this architecture.

3.7.1 *Instruction Syntax*

The x86 assembly has two main syntax branches, *Intel* syntax [The17] (used mostly in Windows) and *AT&T* syntax [bin] (used by GNU tools). Since they are just two different ways of encoding the same underlying semantics, neither one is more expressive than the other and conversion between them is straightforward. Many assemblers support both syntaxes.

3.7.1.1 *AT&T*

- Prefixes: Register names starts with %, constants with $

  `mov $0x400730, %r8`: copies value `0x400730` into the register `%r8`

- Memory operand size: Expected size of operands is specified by the last letter (suffix) of the instruction mnemonic.
  - `b`: byte, 8 bit.
  - `s`: single, 32 bit floating point.
  - `w`: word, 16 bit.
  - `l`: long, 32 bit integer or 64-bit floating point.
  - `q`: quadword, 64 bit.

If at least one operand is a register, the suffix is not needed since the register uniquely determines the operand size.

`mov $0x1, %rax`: `%rax` is 64-bit, there is no need to specify `q`.

`movl $0x1, (%rax)`: needs `l` suffix, since nothing is known about the target address.

- Parameter order: Source first, destination second

  `mov %rdx, %r9`: copies a value from `%rdx` into `%r9`.

- Computed Address: `offset(base, index, scale)` where `offset` and `scale` are constants, while `base` and `index` are registers

  The address is calculated as `offset + base + (index * scale)`

  `mov %edi, -0x4(%rbp)`: Copies value inside `%edi` to address contained in `%rbp` minus `0x4`

  `mov 0x400(,%rax,8), %rcx`: Suppose `%rax` contains 8, then the quadword from address `0x440` is copied into `%rcx`.

3.7.1.2 *Intel*

- Parameter size: Can be specified by prefixing memory operands with:
  - byte ptr: 8 bit.
  - word ptr: 16 bit.
  - dword ptr: 32 bit.
  - qword ptr: 64 bit.

- Parameter order: Destination first, source second. The opposite of AT&T syntax, corresponds better to the typical notation `a = op(b)`

  `mov rdx, r9`: copies a value from `rd9` into `rdx`.

- Computed Address: `[base + index * scale + offset]`

  Unlike AT&T syntax it is obvious how result is computed

  `mov edi, -0x4[rbp]`: Store value inside `edi` on address contained in `rbp` minus `0x4`

  `mov rcx, QWORD PTR [rax * 8 + 0x400]` : Suppose `rax` contains value 8, then `0x440` is copied into `rcx`.

3.7.2 *Registers*

The most commonly used registers are part of a group called General Purpose Registers (GP registers). Registers (register names, not physical registers) in this group are 64-bit (32-bit respectively) wide, with possibility to access lower 8-bit or 16-bit [Amda].

The x86 architecture contains eight GP registers, x86_64 adds another eight. In the assembly the following registers are available to the programmer:

- 64-bit registers:
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP
  - R8, R9, R10, R11, R12, R13, R14, R15
- 32-bit registers:
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
  - R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
- 16-bit registers:
  - AX, BX, CX, DX, DI, SI, BP, SP
  - R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
- 8-bit low-byte registers:
  - AL, BL, CL, DL
  - SIL, DIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, R15B
- 8-bit high-byte registers:
  - AH, BH, CH, DH

Bottom row, if present, represents names available only on amd64. The 64-bit registers are unique, each smaller variant is accessing a part of the corresponding 64-bit register, see Figure 3.6.

SSE (Streaming SIMD Extension) registers form another big group of registers. Name, size and the number of the SSE registers depend on the version of the extension itself. The newest version (avx2) contains thirty-two 512-bit `ZMM` registers. These registers are usually used for vector instructions which execute the same operation on multiple data; therefore, the instructions typically require operands to be of bigger size than a GP register. The floating point arithmetic is usually done with operands in these registers as well. Similar to the GP registers, it is also possible

to access only a part of the register, using different name as shown in Figure 3.7.

A special register is the `rFLAGS` (`eFLAGS` on 32-bit) register, which holds flags. It consists of one control flag and six status flags. Flags are one bit each, i.e. only true or false value. They can be set by both programmer and instructions in the program. For example if integer addition overflows, the overflow flag (`OF`) is set to 1. The `ZF` (zero flag) is set to 1 if the last instruction resulted in the value of 0 which makes the flag important for jump instructions, as they often need to know the result of the previous comparison.

The `RIP` (`EIP`) register is used as instruction pointer.

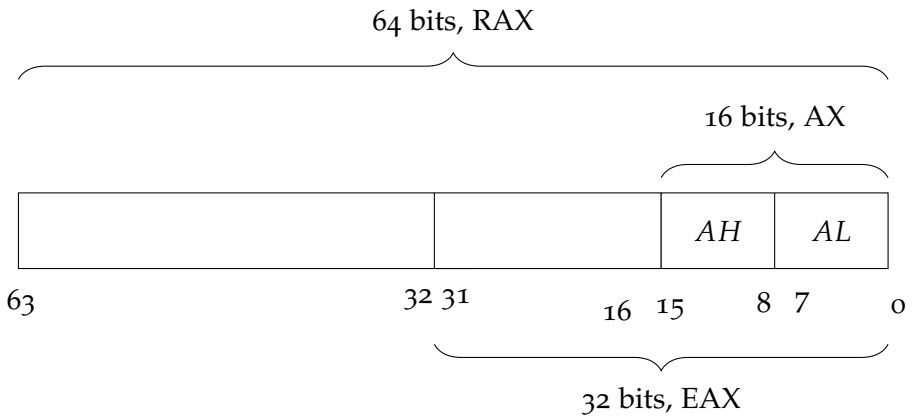

Figure 3.6: Names of accessible parts of GP register

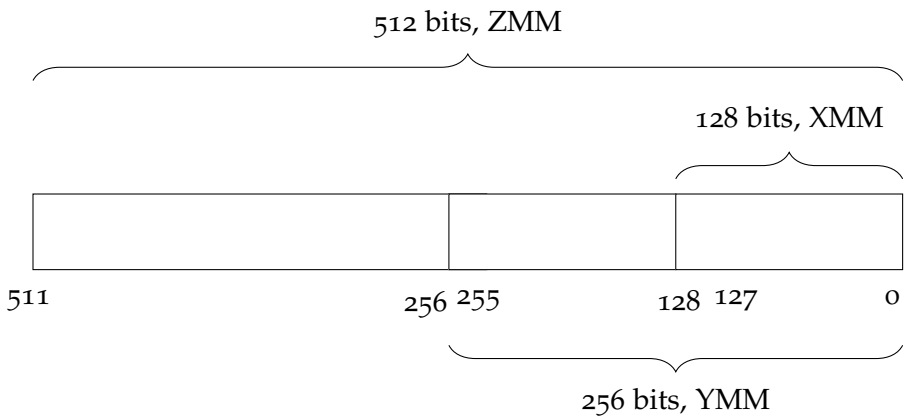

Figure 3.7: Available names of the SSE registers

3.7.3 *Calling convention, Linux*

To determine how to pass the parameters and return values, data types are split into several classes [Jan14]. Some of the basic classes are:

- **Integer**: All integer values that fit into one of the GP registers. Pointers also belong in this class.
- **SSE**: Values that fit into a vector register and floating point values (e.g. C types `float`, `double`).
- **Memory**: For values that are too big to be passed and returned in registers, the stack must be used instead. Anything larger than four quadwords or containing unaligned fields is in this class.

Classes of aggregate and union types are determined separately. If the size of the aggregate is bigger than one quadword, each quadword is classified separately. Class of each quadword depends on the fields contained in it.

Arguments (left to right) are passed in the following order:

- **Integer**: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` and the stack.
- **SSE**: `%xmm0` - `%xmm7` and the stack.
- **Memory**: passed via the stack.

In the case function takes variadic number of arguments, upper boud on the number of used vector registers must be passed in the `%al` register.

The return value is stored in:

- **Integer**: `%rax`, `%rdx`
- **SSE**: `%xmm0`, `%xmm1`.
- **Memory**: The caller provides the space for the return value on its own stack frame. A pointer to this space is passed in as the first argument (in `%rdi`).

Example 3.7.1. *Passing arguments and return values:*

```
// Entire structure is too big
struct T {
  float f_a; float f_b; // together in one SSE
  float f_c; float f_d; // together in one SSE
};
```

```
struct Interval {
  int32_t first; int32_t second; // together in one Integer
};

struct Interval // %rax
foo(
  struct T t,         // %xmm0, %xmm1
  struct Interval a, // %rdi
  uint64_t b,         // %rsi
  char *ptr,          // %rdx
  double d_c,         // %xmm2
  int c,              // %rcx
  short d,            // %r8
  char e,             // %r9
  int64_t f           // stack
);
```

3.8 ELF

Executable and Linkable Format (ELF) [TIS95] is a common file format for storing programs and is used by many Unix distributions. There are several types of objects that are usually encoded using ELF:

- **A Relocatable file** holds code and data suitable for linking (resolving cross-references) with other object files to create an executable or a shared object – a compiler typically produces one object file per translation unit.
- **A Shared object file** holds code and data that can be later dynamically linked (by a dynamic linker) with an executable and other shared objects.
- **An Executable file** holds a program suitable for execution.

The file is made of sections which hold the data and a few structures which store additional information. The beginning of the file contains an *ELF header* that describes the organization of the file. Optional *Program header table* (not needed in a relocatable file) provides information for the system about preparation of the program for execution. Every section has an entry in the *Section table*; its name, size, address in the file and others.

The sections of the ELF file fall into three categories:

- Machine code and static data.

- Tables needed for successful linking.
- Debug info together with other metadata.

Several sections deserve special attention with regard to decompilation:

- `.bss` stores uninitialized data. In the file the section occupies no space (only zeros would be stored), only the size is specified.
- `.data` holds initialized data.
- `.rodata` holds read-only data, later typically loaded into the code segment of memory.
- `.symtab` holds the symbol table. This section may not be always present, to reduce memory footprint of the binary it can be removed.
- `.init` holds executable instructions that are executed when the object is loaded. For executable this means that the code in this section is executed before control flow is passed to the entrypoint (for example the `main` function in C code) of a original program.
- `.fini` holds executable instructions that are executed if the program exits normally or the object is closed.

4

COMPILATION AND LOSS OF ABSTRACTION

In this chapter short overview of compilation process is presented. Nowadays programs are usually written in higher level programming languages and the compilation creates binaries from the source code written in these languages [Aho+06]. Higher level languages are used because they contain abstractions which make them easier to use for programmers. The machine code does not contain abstractions; therefore, after the compilation they must be expressed using sequences of semantically equivalent machine code. Synthesizing the abstractions from the machine code is not a task that can be easily performed (Section 4.2 and Section 4.3).

4.1 COMPILATION

A compiler is a program that translates the input source code, written in a specific programming language (source language), into another programming language (target language) usually machine code (otherwise often called a 'transpiler'). Translation between languages can be done for example to benefit from compiler infrastructure (optimizations, machine code generation and others) available for the target language.

4.1.1 *Front-end*

First several steps of a compiler are focused on checking the syntactic correctness of the program, while building an internal representation required in later phases. Incorrectly formed programs are rejected and appropriate warnings and errors are generated.

The first step of compilation (lexical analysis) is splitting the input source code into tokens or lexemes. Tokens can be sorted into several categories: identifiers (names), operators, constant numbers and string literals. Categories may differ per programming language.

Validy check of units is also part of lexical analysis. Only the validity of tokens as units is checked, incorrect sequences of tokens are caught in later stages of compilation. Syntax of tokens is typically regular language, therefore a simple analyzer based on finite automata is often sufficient [MY60].

Syntax analysis identifies the syntactic structure of the program. The list of tokens created in the lexical analysis is transformed into a parse tree, which mirrors the context-free grammar of the language. As the tree is constructed, the compiler performs syntactic checks to determine whether the order of token matches the grammar.

The last phase is semantic analysis. It performs additional checks and adds information to the syntax tree. This typically includes type checking (if rules of the language allow it, implicit type conversions are added), a symbol table is being built, checks are being performed to decide whether local variables are defined before used and others based on the chosen compiler

As every language has specific syntax, every compiler needs to provide its own frontend. It is possible to re-use middle-end and backend of a different compiler that already exist, if the frontend produces a corresponding intermediate representation.

4.1.2 *Intermediate representation & Optimization*

After syntax and semantic analysis of source program, many compilers generate a low-level machine-like intermediate representation. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target code (LLVM bitcode [LLV19] is an example of such representation). A compiler may construct one or more intermediate representations, which can have multiple forms.

After the generation of the intermediate representation, a machine agnostic code optimization phase begins. It tries to improve the intermediate code so that better target code can be generated. Usually better means faster, but sometimes shorter code (in case memory is limited) or code that consumes less power (embedded device) may be desired. The amount and types of optimization available differs from compiler to compiler.

4.1.3 *Code Generation*

The last phase of compilation maps the intermediate representation to the target language. Operations in the intermediate representation are replaced with a sequence of instructions of the target architecture that perform the same task.

Register allocation is performed, for each virtual register of the intermediate representation it is decided, whether it will reside in a register or memory. Efficient utilization of available registers is the main focus.

Allocation depends on the target machine, since each processor contains different set of registers [Aho+06].

4.2 ABSTRACTION

Programming languages can be split into categories based on how much abstraction (from the actual machine code) and features they provide. It is easier to move down the abstraction ladder (replace abstractions with simpler ones) than up (replace sequence of statements or instructions in the lower language with a suitable abstraction).

4.2.1 *Architecture Specific*

Bottom of the abstraction ladder is the machine code. It consists of individual instructions encoded in a binary format optimized for simplicity of hardware decoding and for compactness. The encoding itself depends on the processor the machine code is supposed to be run on. There are no symbolic labels, everything is represented by addresses and offsets to them.

It is the only language a computer can execute without any previous transformation. Although programs are not written in machine code, many programs end up being represented that way, so they can actually be run.

Assembly language is one level above the machine code. It abstracts from addresses and offsets, replacing them by symbolic labels. Instructions are encoded in human readable format.

Although it was originally designed for humans it is rarely used nowadays, since there are often better alternatives. For more detailed description see Chapter 3.

4.2.2 *Lower Level Languages*

Lower level languages are typically compiled (the C language is an example), therefore require another program to transform programs written in them to the machine code. These programs are called compilers and the process is called compilation (Section 4.1).

They add another layer of abstraction on top of assembly. The specifics of the hardware architecture are no longer critical to a program; registers and explicit stack operations are no longer needed and are abstracted by local variables. More complex types are present (structures, unions). The more general syntax of the language replaces the instruction mnemonics

(operator + can be used to add two values instead of an architecture specific `add` instruction). Control flow is more structured and visible, compared to explicit jumps and calls.

Even though they are more abstract than assembly, in practice they are used mostly for OS-level programs. These language are still partially platform dependent. Architecture of the processor is abstracted away, however specifics of the target operating system remain.

4.2.3 *Higher Level Languages*

Higher level languages provide even more abstraction. A better type system is typically present. Late dispatch, lexical closure, generics, overloaded functions and other constructs are often part of the language. The full set of features depends on the language.

The top of the ladder are interpreted languages. These languages are typically not compiled into machine code but are instead executed in an interpreter (virtual machine). Similar to a compiler, the interpreter first checks the syntax of the input program, but then instead of optimizing the code and generating a program in the target language, the program is run in environment of the interpreter. Interpreter knows how the semantic of the language is defined and on the fly "interprets" the source program.

4.3 LOSS OF ABSTRACTION

As the code is compiled into a lower target language, abstractions provided by the source language are lost. Their semantics remain, since compilation does not alter the semantics of the program, but they are expressed by means available to the target language.

This section shows examples of how abstraction is lost during typical compilation of a C program. Source language (C) is transformed into intermediate representation used by compiler (LLVM bitcode, see Chapter 2), which serves as an input for assembly or machine code generation.

4.3.1 *C → LLVM*

Most obvious loss is the explicit control flow represented by `if` and the loops. In the bitcode, there is only branching which makes detection of loops harder for human reader than explicit statements in the C code.

Although LLVM has its own type system, resulting types in the bitcode do not have to exactly match the original types in the source program.

Example 4.3.1. *Consider following data structure and function which uses it:*

```
struct Interval {
  int32_t begin;
  int32_t end;
};

struct Interval sum( struct Interval lhs, uint32_t factor );
```

The corresponding prototype of the function in the bitcode may not include the `struct Interval` *at all. The compiler may change the original type of the argument to some other equivalent type, to prepare for better machine code generation.*

```
%struct.Interval = type { i32, i32 }
declare i64 @sum(i64, i32)
```

In the bitcode `Interval` *is represented as only one packed argument, since it corresponds to the calling convention of the target architecture and operating system.*

While structures may be lost in some cases, unions are not present in the bitcode at all, since LLVM type system has no equal type. They are implemented as pointers with the `bitcast` instruction (the `bitcast` instruction changes the type of a pointer).

Once optimization passes are taken into account, some additional information is lost. Local variables may be optimized into virtual registers and not recognizable; functions may be inlined into their callers, losing information about function calls; instructions may be reordered.

4.3.2 *LLVM → Machine Code*

The transformation of bitcode to machine code leaves most of the control flow structure intact.

Machine code has no explicit types and specifically no aggregate types. The type information can only be guessed from the operands and instructions, but even then it does not provide precise information about the original type used in source program. Aggregate type in the bitcode represents some amount of memory and its field members represent its internal structure. The memory itself remains in assembly as well (may be slightly bigger thanks to alignment) but information about the internal structure (field members) is not explicitly stated. Only deductions based on instructions and the size of their operands provide some insight

on how did original structure looked like, but it cannot be reconstructed accurately [TDC10] [EW04].

Some of the basic blocks are entry blocks (targets of the `call` instruction) and some blocks contain the `return` instruction. All blocks lying on the execution path from the entry block to the block with `return` instruction (without following the `call` instruction) form a subroutine. This partially corresponds to the functions from higher level languages – they can be called and they return. However type of the subroutine is no longer present as an easily retrievable information. While in the higher level languages the function definition typically must contain the type, the machine code does not have this requirement. The type of the subroutine is still implicitly present, the subroutine itself retrieves its arguments from the correct registers and callers store them properly as well.

For functions that abide the calling convention (functions with external linkage) the arguments can be partially retrieved by an analysis of the call sites. Functions that are used only internally do not have to abide the calling convention, but those produced by compiler usually do.

Example 4.3.2. *Consider the following C function that calls another function in a simple loop:*

```
void printer( int size, int base ) {
  for ( int i = 0; i < size; ++i ) {
      func( i + base );
    }
}
```

The C code is relatively short and it can be easily seen what is happening. Function type is explicitly stated and so is the control flow. Note that local variable `i` *is present.*

```
define void @printer(i32, i32) local_unnamed_addr #0 {
  %3 = icmp sgt i32 %0, 0
  br i1 %3, label %4, label %5
; <label>:4:                                       ; preds = %2
  br label %6
; <label>:5:                                       ; preds = %6, %2
  ret void
; <label>:6:                                       ; preds = %4, %6
  %7 = phi i32 [ %9, %6 ], [ 0, %4 ]
  %8 = add nsw i32 %7, %1
  tail call void @func(i32 %8) #2
  %9 = add nuw nsw i32 %7, 1
  %10 = icmp eq i32 %9, %0
```

```
  br i1 %10, label %5, label %6
}
```

Bitcode is more verbose than the C code it originates from. The function type is still present and unchanged, only sizes of the integer types are explicitly stated. However from the body of the function, it cannot be easily seen that it contained a loop. Local variable `i` *is no longer present, it was optimized away.*

```
printer:
  push   %rbp
  push   %rbx
  push   %rax
  mov    %esi,%ebx
  mov    %edi,%ebp
  test   %ebp,%ebp
  jle    .epilogue
  nopl   $0x0(%rax,%rax,$0x1)
.body:
  mov    %ebx,%edi
  callq  func
  inc    %ebx
  dec    %ebp
  jne    .body
.epilogue:
  add    $0x8,%rsp
  pop    %rbx
  pop    %rbp
  retq
```

Assembly code has structure similar to that of the bitcode. It contains fewer basic blocks, since in the bitcode, every basic block must end with a terminator, which is not the case in assembly. The beginning of the subroutine can be easily seen, though type information is not present. Situation is similar for the `call` *instruction; the name of the subroutine is specified, but there is no information about the arguments or the potential returned value.*

In this specific case, with the knowledge of the underlying platform (amd64 and Linux) and corresponding calling convention, it can be seen that the subroutine probably has only two arguments and no return value: `%esi` *and* `%edi` *are used and* `%rdx` *is not, neither any other location that argument can be passed in. The subroutine does not store anything in any register that could contain the return value, therefore it probably does not return anything.*

5

MCSEMA

Lifting a binary to LLVM bitcode is a process that creates bitcode with semantics equivalent to the original binary.

There are several reasons why lift to LLVM bitcode can be desired:

- **Binary patching and modification:** Lifting the binary into bitcode allows modification of the program. Functionality can be added or removed, safety checks can be inserted or some functions can be rewritten altogether. After the modifications are done, the result can be easily compiled into a new binary which contain all the changes.

- **Recompiling for different architectures:** Programs compiled for one architecture and operating system usually cannot be run on a different architecture. Lifting the binary allows the result to be recompiled again with the desired architecture as the target.

- **One set of tools:** Analysis of LLVM bitcode is purpose of many tools available – static analyzers, model checkers, etc. By lifting the binary these tools can be run on previously unavailable targets.

- **Analysis of the binary itself:** Usually the analyzer tools analyze the source code, which can be very different than the compiled binary. Compiler optimizations and code generation can alter the implementation significantly, while preserving the original semantics. Lifting the program allows analysis of the code that is actually being run, instead of the original which may be altered in the compilation process. When analyzing the program, sometimes only binary is available; lifting it allows analysis on previously unavailable targets.

The lift process may however introduced its own problems, since the process does not exactly re-create the original – errors may be inserted during the lift and behavior may be added or lost. An optimal result is a bitcode that is identical to the bitcode produced as an intermediate representation during the original compilation. Unfortunately, this is not possible (for more details see Section 4.3).

5.1 MCSEMA ARCHITECTURE

McSema [Din+19] is a tool that lifts (translates) executable binaries from machine code into LLVM bitcode.

McSema is split into two parts: the frontend and the backend. The frontend is a lightweight program that uses an external disassembler to retrieve all information about the binary needed to successfully decompile the program.

The backend takes the output of the frontend as input (binary is no longer needed) and creates the corresponding bitcode. The backend uses Remill library to lift the individual instructions.

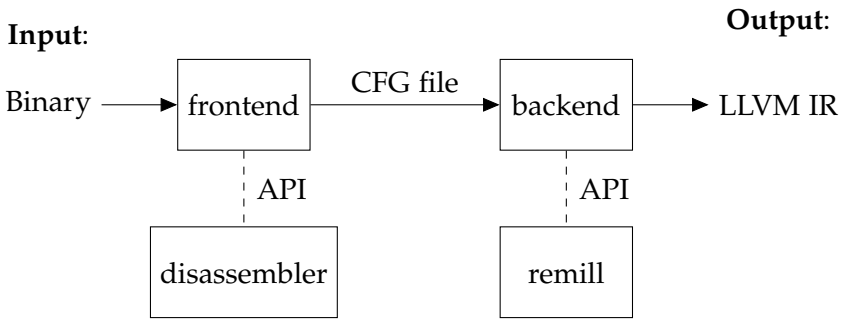

Figure 5.1: McSema components

First step is to retrieve information about the program encoded in the binary. McSema does not provide its own disassembler, therefore requires external tool to do the job. Currently supported disassemblers are IDA Pro [Hex19] and partially Binary Ninja [Vec19].

Advantage of using third party tools is that there is no need to implement a fully fledged disassembler which is not an easy task. Others have done it already and using their tools saves time.

To support a disassembler small program that uses its API has to be written, since the backend expects information about the binary in the specific format (CFG file). Since the binaries are very dependent on target architecture, both the frontend and the backend require as a parameters the operating system and the architecture of the processor for which the original binary was compiled.

5.2 CFG FILE

To support multiple disassemblers, which often provide bindings to different languages, common message format (CFG file) between the frontend and backend is defined.

Each binary is contained in one CFG file. Internal structure of the CFG file can be described using grammar in EBNF (Extended Backus-Naur form) [Wir96]. Figure 5.2 shows such grammar.

```
<Module> := { <Segment> | <External Variable> | <External Function> |
             <Function>| <Global Variable> };

<Segment> := address, name, data, is_read_only, is_thread_local,
            { <Data Reference> };
<Data Reference> := address, width,
                    target_address, target_name, target_is_code;

<External Variable> := name, address, size, is_weak, is_thread_local
<External Function> := <Calling Convention>, name, address,
                       no_return, argument_count, is_weak;

<Function> := address, { <Block> }, is_entrypoint, [name],
              [ <Stack Variable> ];
<Block> := address, { <Instruction> }, { successor_addresses };
<Instruction> := address, bytes, { <Code Reference> }, [local_noreturn];
<Code Reference> := <Location>, <Operand Type>, address;
<Location> := "internal" | "external";
<Operand Type> := "immediate" | "memory" | "displacement" |
                  "control flow" | "offset table";

<Stack Variable> := names, size, sp_offset,
                     { <Instruction Reference> };
<Instruction Reference> := instruction_address, offset;
<Global Variable> := address, name, size;
```

Figure 5.2: Simplified EBNF Grammar of the CFG file

5.2.1 *References*

A program is a sequence of bytes, where each byte is assigned an address. When one part of the program needs to refer to another part of the same program, it does so using such an address, which is then stored using a machine-specific encoding. The portion of the program that encodes an address in this form is called a *reference*. The address that is encoded by the reference is called the *reference target*. The reference itself, also being part of the program, is stored at some *reference address*.

References can be thought of as initialized pointers in the binary.

Example 5.2.1. *References are often result of a pointers in the source code, consider the following C code:*

```
char *global = "I am global string";
char **ptr_to_global = &global;
char *hardcoded_ptr = 0x123456;
```

The generated data section after compilation contains all the pointers from the code above, which are referring to some other parts of the binary.

```
Hex dump of section '.data':
 0x00601020 00000000 00000000 00000000 00000000 ................
 0x00601030 d4054000 00000000 30106000 00000000 ..@.....0.`.....
 0x00601040 56341200 00000000                   V4......
```

Address `0x601030` corresponds to `global` (its target is in a different section). The `ptr_to_global` is stored at `0x601038` and its value is the address of the `global` (little-endian encoding). Both of these addresses are references.

There are two types of references. The type of a reference depends on the location it is used in:

- **Code references**, represented by `<Code Reference>`. References of this type are found in operands of instructions.

  Whether constant operands represent are a reference depends on the type of the binary. If the binary is position independent, every reference to the another part of the binary is in form of an offset from the instruction pointer therefore constants are never a reference. Position dependent code is more ambiguous, as constants operands which represent addresses in the binary look the same as any other constant number.

  ```
  402b15:  be c0 81 65 00     mov     $0x6581c0,%esi
  ```

  The best guess is that `0x6581c0` is referring to the address, since it points into one of the sections of the binary and is properly aligned, but there is no way to be certain that it does not simply represent the number `0x6581c0`.

- **Data references**, represented by `<Data Reference>`.

  References in data sections are harder to detect, since except for alignment (which does not have to always be present) they have no guaranteed structure. Data sections are typically filled with constants used in the original source code, for example constant integers or string literals.

  In the case of data references, position independent code makes their detection more difficult. Addresses in the binary are usually

lower (since they often start at zero) which makes their overlaps with constants more frequent.

```
0x00013410 20202d56 2c202d2d 76657273 696f6e20   -V, --version
0x00013420 20202020 64697370 6c617920 76657273     display vers
0x00013430 696f6e20 6e756d62 65720000 00000000 ion number......
```

The address 0x13438 contains quadword `0x6572` which can be interpreted either as the numerical value `0x7265` (little-endian encoding) that can easily be an address of a function, or as two letters: `e` and `r`.

Unfortunately, there is no correct way to deal with values that may be either references or constant values. Decision in favor of the constants can lead to a control flow error, such as indirect call to a fixed address which results in a segmentation fault if the program is ever recompiled.

Favoring the reference can also lead to an error, for example comparison to an address of a function instead of some fixed constant, will result in the lifted program taking different branch than the original one.

5.2.2 *Segments*

One `<Segment>` corresponds to a section (or its contiguous part) in the original binary. There are several attributes of the segment that deserve special attention.

`data` represents verbatim copy of all data in the section, with the exceptions of the sections which contain relocation tables (`.got` for example).

In these section relocation are first applied – it results in a different data than the section originally contained. Semantics are however preserved, addresses that stored the addresses of the external functions in the original section, contain addresses of the same functions after relocations are computed.

`<Data Reference>` represents a reference stored in the section. All references must be specified; otherwise the lift may fail. The most important attributes are the start address and width of the reference, typically size of a pointer. Whether the reference is targeting a code (function pointer) should be specified as well, even though it can be often easily deduced.

Section can be split into multiple `<Segment>` entries with the same name. The main idea is to split the section into smaller chunks that are

independent from each other (For example segment per global variable). This allows finer grained bitcode to be produced.

Incorrect split may result in the bitcode not working as the original binary did. For example consider section containing two strings which is split into two segments, each containing one of the strings. If an instruction tries to access second string via pointer to the first one, the lifted bitcode will behave differently, since the string are no longer guaranteed to be contiguous in the memory.

5.2.3 *Functions*

The `<Function>` in the CFG file corresponds to a sequence of basic blocks in the assembly starting with entry basic block (known beginning of a function or target of a call) and all other basic blocks that can be reached from it – the `call` instructions are not followed. Every target of a `call` instruction that can be statically determined should be marked and included as a function.

While list of basic blocks (`<Basic Block>`) is the most important characteristic, there are other properties of the function that should be specified.

`is_entrypoint` specifies whether some external function may call the function either directly or indirectly.

`<Local Variable>` represents list of stack variables if they can be determined, which usually depends on the binary containing debug info. Without debug info, it is not possible to recover local variables accurately; however, some heuristics are available [BR07] [LAB11].

5.2.4 *Basic Blocks & Instructions*

A basic block contains list of all instructions present and a list of addresses of its successors. There is usually one (unconditional jump or no control flow instruction) or two successors (conditional jump).

In the case of an indirect jump however, if targets can be deduced (jump table) they should be included in the successor list. If the targets cannot be decided, the basic block has no successor originating from that jump.

The instruction entry must contain bytes of the instruction (`bytes`) – they are later used as an input for Remill, which decodes them itself. In case one of the operands of the instruction is a reference to some other object in the binary, the information about reference should be included (`<Code Reference>`).

5.2.5 *Externals*

Symbols which do not have definitions in the binary are called external. A binary usually contains several external variables and functions.

For external variables (`<External Variable>`) it means that their addresses is in some other object file that will be dynamically linked later. The address of the relocation of the external variable should be used as its `address`. In case of the relocation with address in a section of the original binary, a corresponding reference should be added to that section. Typically this results in a reference at some address with its own address as a target.

Opposite to external variables, external functions (`<External Function>`) do not use the address of the relocation as their own. First relocation is calculated – every external function is given an address outside of the original binary, therefore it cannot collide with an object present there. The new address is used as an address of the function and all the references which targets this function use it as a target. Global Offset Table (GOT) typically contains references to all external functions, therefore these must be changed to target the new addresses. The `data` (actual bytes) of the segment should also be updated.

McSema has its own file which contains definition of all possible external functions (definition file); their names, arguments number, calling convention and whether they return (all these attributes must be specified in the CFG file). This option is not very convenient as for every program that uses shared libraries, all used external functions that are not already present in the definition file must be added. The CFG file still requires these fields to be filled, although during the lift user is allowed to add a more accurate description overwriting the one specified in the CFG file.

For each instruction that targets the external function (common example is the `call` instruction) there are two ways how to resolve the reference, since its target is an entry in the GOT:

- Most correct option is to let the instruction target the entry, which results in a precise simulation of the instruction.
- It is possible to ignore the original reference target and set it to the external function directly. This option may result in an incorrect behavior – the table is not read-only and some instruction could change an entry to a different function during the execution. However, binaries produced by the compiler do not rewrite the table after the relocations are resolved, therefore this option typically works.

5.3 STATE

Main idea behind McSema is a simulation of the original binary. To successfully simulate any program, simulation of the environment it runs in is mandatory. For the compiled binary environment means the registers of the target processor.

There is no explicit stack by default. Instructions operating on the stack must retrieve it's current top from the appropriate member field in the `State` structure.

To simulate state of the processor a structure called `State` is used. Member fields of the `State` structure correspond to every register (register name, not a physical register) that can be used during the execution of the program. Special definition must be used for each supported architecture.

The `State` structures should maintain several properties, so that emulation is accurate.

- They should have the same size across architecture revisions and generations. Maintaining this property allows combining separately lifted bitcodes. For example one file with avx support and second without, or x86_32 and amd64.
- The structures should be designed in a way to prevent certain compiler optimizations that would hinder the lifted bitcode such as, load and store coalescing.
- Every register available in the processor should be a member field with correct name and size.
- Conversion between the `State` structure and actual running processor should be easy.

5.4 REMILL

Remill is a standalone library focused on lifting machine code instructions to LLVM bitcode. To lift an instruction, its representation as sequence of bytes should be passed to Remill.

Flowchart that shows the lift process of an instruction is presented in Figure 5.3. The instruction is first decoded and then translated to remill bitcode, which is essentially LLVM bitcode extended with:

Intrinsics are operations used in the bitcode that are not part of LLVM language. Their realizations are left to the user.

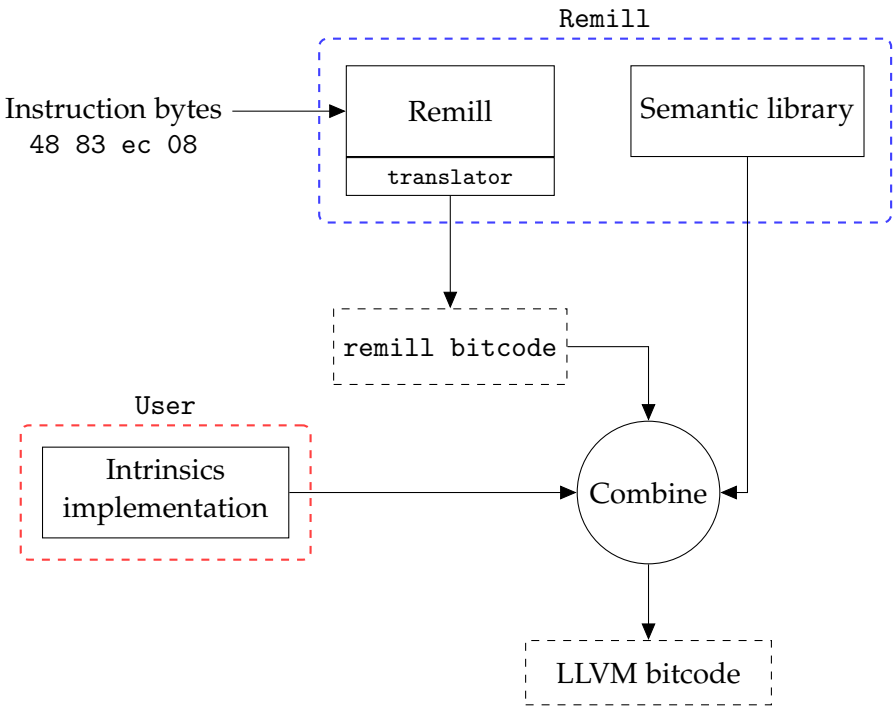

Figure 5.3: Lift of an instruction using Remill.

Semantic functions which implement the semantics of instructions, i.e. their effect on the `State` structure. Remill (the remill semantic library) provides an extensive collection of implementations of such functions.

`%struct.State *`: pointer to the `State` structure which holds all registers.

`%struct.Memory *`: pointer to the opaque `Memory` structure. It ensures ordering of memory operations.

To transform remill bitcode into LLVM bitcode, definitions of semantic functions and realizations of the intrinsics must be provided.

Example 5.4.1. *Remill instrinsics which read and write 8-bits of memory:*

```
declare i8 @__remill_read_memory_8(%struct.Memory*, i64 %addr)
declare %struct.Memory* @__remill_write_memory_8(
  %struct.Memory*, i64 %addr, i8 %value)
```

Instrinsic for indirect function call:

```
declare %struct.Memory* @__remill_function_call(
  %struct.State* , i64 %addr, %struct.Memory*)
```

McSema, as a user of remill, provides realization for these instrinsics. As it tries to simulate the behavior of the program, simple `load` *instruction is enough to replace the read intrinsic and* `store` *instruction to replace write intrinsic. Indirect function call is implement using more complex mechanism (see Section 5.6.1).*

5.4.1 *Implementation of Semantic Functions*

First two arguments of semantic function are independent from the instruction, as they provide access to the simulated environment: pointer to the `State` structure and pointer to the `Memory` structure. The remaining arguments depend on the instruction itself and their number usually corresponds to the number of arguments of the original instruction.

Instruction semantic is implemented using C++ and later compiled into LLVM bitcode. Using C++ templates and macros makes it easier to expand and maintain the functions, compared to using LLVM API, which would be more verbose and harder to read. Using C++ metaprogramming it is also easier to generate more functions with one implementation in a clear way.

```
template <typename D, typename S1, typename S2>
DEF_SEM(SUB, D dst, S1 src1, S2 src2) {
  auto lhs = Read(src1);  // Read value form source
  auto rhs = Read(src2);
  auto sum = USub(lhs, rhs); // Unsigned substraction
  WriteZExt(dst, sum); // Write into dst
                       // zero extend if types do not match
  WriteFlagsAddSub<tag_sub>(state, lhs, rhs, sum); // Update ArithFlags
  return memory;
}
```

Figure 5.4: Definition of the semantic function of the x86 `sub` instruction

`DEF_SEM` is a macro for defining a new instruction. First argument is the name, typically the same as the name of the instruction, and the rest is variable number of arguments the semantic function uses. Arguments are templated and named accordingly to the usage inside the definition (by convention), sources (`src`) are being read from and destinations (`dst`) written to.

The body of the definition implements the same effect on the `State` structure as had the original instruction on the processor. Remill provides several convenience functions, operators, to implement the body

in a way that is easily readable. Operators help the abstraction, since they provide uniform way to work with arguments. In Figure 5.4 for example, the `Read` operator behaves correctly in both possible cases of argument type – constant or register – without need to explicitly specify them. All functions called in Figure 5.4 are operators.

All the arguments must already be resolved when calling the function, therefore semantic functions themselves are not enough to lift an entire binary, since there are many references that must be already mapped to corresponding bitcode objects to be lifted properly.

Example 5.4.2. *Lift process of a simple instruction. Consider the following instruction:*

```
48 83 ec 20     sub  0x20,%rsp
```

First, Remill uses the decoder to determine type and operands of the instruction. Then the corresponding semantic function is chosen (there are multiple versions, depending on operand types) and a call is generated.

```
%rsp_val = load i64, i64* %RSP
%new_mem = call %struct.Memory* @SUB<i64*, i64, i64>(
    %struct.Memory* %mem, %struct.State* %0,
    i64* %RSP, i64 %rsp_val, i64 32)
```

5.5 MCSEMA

As the Remill library focuses on lifting separate instructions without any context, it is not enough on its own to lift an entire binary. McSema handles the re-creation of the entire context binary was intended to be executed with (sections, global variables, externals) and uses Remill to lift the individual instructions.

5.5.1 *Segments*

Each `<Segment>` from the CFG file is lifted as global variable in the bitcode, which corresponds to its scope in the original binary. Since their size is constant they can be represented as arrays. The global variable is initialized with the data of the segment, then all references are replaced. For each reference, its target address is replaced by a pointer to the corresponding bitcode object. Important semantic remains, if an instruction dereferences the address, the result is the bitcode representation of the same object as in the original binary.

Example 5.5.1. *Consider the following sections:*

```
 Hex dump of section '.rodata':
 0x004005b0 01000200 4920616d 20676c6f 62616c20 ....I am global
 0x004005c0 73747269 6e6700 string.

Hex dump of section '.data':
 0x00601020 00000000 00000000 00000000 00000000 ................
 0x00601030 d4054000 00000000 30106000 00000000 ..@.....0.`.....
```

The `.data` *section contains two references – one starting at* `0x601030` *targeting* `0x4005b4` *and second at* `0x601038` *targeting the previous one. The corresponding global variables in the bitcode (names are shortened compared to real output):*

```
@seg_rodata = internal constant %rodata_type
  <{ [23 x i8] c"\01\00\02\00I am global string\00" }>

@seg_data = internal global %data_type <{ [16 x i8] zeroinitializer,
 i64 add (i64 ptrtoint (%rodata_type* @seg_rodata to i64), i64 4),
 i64 add (i64 ptrtoint (%data_type* @seg_data to i64), i64 16) }>
```

The `.rodata` *section contains the same bytes wrapped in an array of correct length. The* `constant` *keyword marks it as non-mutable, since the section was read-only in the original binary.*
The variable representing `.data` *section can be split into two parts. First is zero initialized array which corresponds to the first sixteen bytes of the original. Second are the two references, which are no longer represented as an addresses (which served as identification only in the original binary) but as pointers into the bitcode arrays that represent the sections.*

5.5.2 *Functions*

For each `<Function>` from the CFG file there are at least two lifted functions (exact number is dependent on the chosen lift mode) in the resulting bitcode:

- The function which simulates the original one. The type of this function is artificial and it is the same as the types of all other helper functions used by McSema.

  ```
  %struct.Memory* (%struct.State*, i64, %struct.Memory*)
  ```

  First argument is a pointer to the `State` structure, second is current value of a program counter and last is a pointer to the `Memory` structure, which together with return type allows ordering of memory operations if required.

```
define %struct.Memory* @sub_400520_main(
  %struct.State*, i64, %struct.Memory*)
```

In the example above typical naming convention is shown. It consists of `sub_` prefix followed by a hexadecimal representation of the original address of the function and lastly the actual name is appended.

- Wrapper function (indirection wrapper), serving as a gateway for the control flow not originating in the lifted bitcode or indirect calls. Actual implementation is dependent on the chosen lift mode, but it usually only stores its arguments into the `State` structure and calls a function (can be another wrapper). Each time the address of the original function is target of a reference, pointer to the corresponding indirection wrapper is used. It has the name of the original function.

The actual lift process is straightforward, for each `<Basic Block>` in the CFG file the corresponding basic block in the function is created. The block is then filled with its instructions, lifted by Remill's semantic functions.

Example 5.5.2. *Lift of a simple function using the semantic functions. Consider the following simple function* `foo`*:*

```
foo:
  movq $0x42, (%rsi) # 48 c7 06 42 00 00 00
  add $0x1, %rdi     # 48 83 c7 01
  callq boo          # e8 e8 ff ff ff
  retq               # c3
```

The following pseudocode highlights the idea behind the lift of the whole function:

```
Memory *sub_400570_foo(State *state, int64_t pc, Memory *memory) {
  auto *rip = state->gpr.rip;
  auto *rsi = state->grp.rsi;
  auto *rdi = state->gpr.rdi;

  // movq 0x42, (%rsi)
  *rip += 7;   // add size of the instruction
  memory = MOV<I64, R64W>(memory, state, 0x42, *rsi);

  // add 0x1, %rdi
  *rip += 4;
  memory = ADD<I64, R64, R64W>(memory, state, 0x1, *rdi, rdi);
```

```
  // callq boo
  *rip += 5;
  // simulation of the effect of a call instruction on the State
  memory = CALL<I64>(memory, state, address_of_boo_in_binary, *rip);
  // actual call
  memory = sub_400568_boo(memory, rip, state);

  // retq
  *rip += 1;
  memory = RET(memory, state);

  return memory;
}
```

If the `<Local variable>` are specified in the CFG file, McSema can lift them into separate `alloca` instructions instead of anonymous offsets in the stack. All instructions that use the variable (`<Instruction Reference>`) are also changed; instead of a pointer to offset in the stack they use the pointer to the newly allocated memory. The resulting bitcode is better, since it is more similar to the original code. Subsequent analysis can also be more precise as well, since it can work smaller parts of memory rather than one big stack.

5.5.3 *External Functions*

List of a external functions used in the binary is not hard to obtain. However, the function declaration in the bitcode needs to have a type and as mentioned earlier (Section 4.3), there is no easy and correct way to obtain the type from the binary itself.

By default, a declaration of the external function is created based on number of arguments passed in the CFG file; return type is `i64` and all the arguments have type `i64` as well. Type `i64` is chosen because the most common types of arguments of a function are in a class that is passed in the same registers as `i64`.

This solution is not ideal, although it is enough for the code to be recompiled again and work as intended. Functions that take a variable number of arguments have their argument count set to sixteen, which is usually a safe approximation. If the function would be called with more arguments, recompiled code would not work.

5.5.4 *Call Reconstruction*

Providing LLVM declarations of external functions allows McSema to try to reconstruct external calls. Every call of an external function is done in a specific way defined by the calling convention of the target architecture; therefore, the location of every argument is known (with the exception of functions with a variable number of arguments).

Example 5.5.3. *Consider simple C function* `puts` *and a slightly simplified bitcode reconstructing call to it.*

```
declare i32 @puts(i8*) #16

define %struct.Memory* @ext_puts(
  %struct.State* %state, i64 %pc, %struct.Memory* %memory) {

  ; First basic block defines virtual registers with names
  ; corresponding to the registers as pointers
  ; to the correct member fields of %state

  %RDI_value = load i8*, i8** %RDI  ; load current value of %RDI
  ; it must be casted properly
  ; Recreate the return
  %RSP_value = load i64, i64* %RSP, align 8
  %0 = inttoptr i64 %RSP_value to i64*
  %return_address = load i64, i64* %0
  store i64 %return_address, i64* %RIP, align 8

  %RSP_pop = add i64 %RSP_value, 8
  store i64 %RSP_pop, i64* %RSP, align 8

  ; Actual call
  %puts_return = tail call i32 @puts(i8* %RDI_value)
  %c_return = zext i32 %puts_return to i64
  store i64 %c_return, i64* %RAX, align 8 ; store into return register
  ret %struct.Memory* %memory
}
```

As with all other calls when lifted, two sequences of instructions are added to the function. First is the call to the semantic function of the `call` instruction, which typically pushes the current instruction pointer to the stack, and then the actual call. If the call is to another internal function, these two operations are enough. However if the call is directly to another bitcode function (without wrappers), a problem arises, since

there would be no `ret` instruction matching the `call` done by the caller and the stack pointer would end up with incorrect value. The wrapper that re-creates the call (naming convention `ext_` + address + name) has to simulate the `ret` instruction explicitly, as can be seen in the Example 5.5.3.

5.6 LIFT MODES

Calls to external functions make lifting more complicated. Transfer of control flow between internal functions is without problems, they all operate on the same `State` structure (*lifted context*). The calls work implicitly – the original machine code placed arguments in the correct locations, the callee looked for them in the same place. The lifted bitcode inherits this property, as the caller puts the arguments into the same registers, difference is only that the registers are now member fields of the `State` structure. Symmetrically, the callee also retrieves the arguments from the correct member fields of the `State` structure.

This property however does not hold when the callee or the caller is an external function (*native context*), as it has no concept of the context in which lifted code operates; therefore, a layer between the two must be implemented. This is not the case if the external function has proper bitcode type supplied as the call can be reconstructed.

Currently, there are two different implementation of this layer, based on the intended usage of the lifted code.

5.6.1 *Recompilation mode*

By default the layer between *native* and *lifted* context is implemented in a way that favors recompilation rather than analysis of the bitcode. The main idea is that the *native* and the *lifted* context are synchronized each time context is swapped. Synchronization also happens as the first thing in the `main` function.

Direct transfer of control is shown in Figure 5.5. Compared to the reconstructed call, the wrapper around external function does not have to simulate callee's return, since the last instruction of the callee before returning is typically `ret` instruction, therefore the stack pointer is properly adjusted.

Aside from direct calls, another transfer of control flow is an external function calling a function in the lifted bitcode. Typical example is the `qsort` function calling the comparison callback. Since the external function calls other functions normally, i.e. without knowledge of the

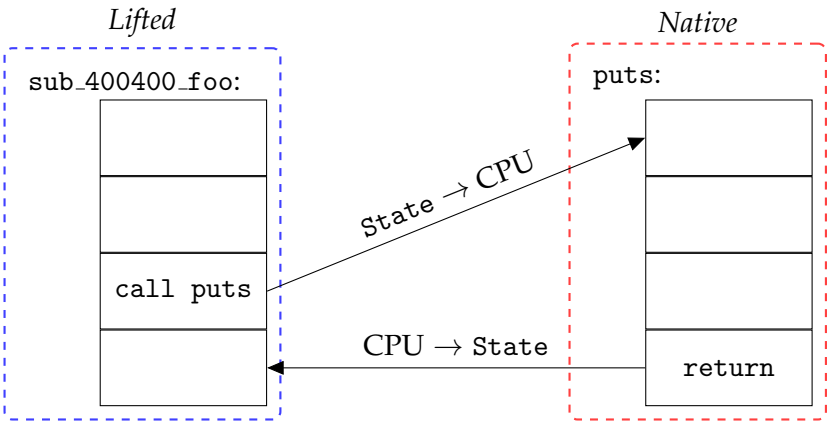

Figure 5.5: Synchronization of contexts

lifted context, the indirection wrapper must ensure synchronization of contexts.

A scheme of this process is shown in the Figure 5.6. Two more helper subroutines are used. First (`attach_call`) is called by the indirection wrapper and performs following:

- Copies the content of the processor into the `State` structure.
- Sets the return address on the stack to the beginning of the `detach_call_ret` subroutine.
- Jumps to the first instruction of the callee.

Altering the control flow allows synchronization other way, the `State` structure is copied into the processor (`detach_call_ret`) before control is handed back to the original caller after the callee is fully executed.

Indirect control flow is handled in the same way, only difference being that the *lifted* context is re-entered from itself via specific dispatch function (`qsort` represents it in Figure 5.6).

Subroutines that ensure the synchronization cannot be written using only LLVM bitcode instructions, since the subroutines explicitly manipulate CPU registers. As a results, they are implemented in assembly and later linked with the lifted bitcode during recompilation.

Using assembly subroutines for synchronization ensures that the recompiled code always works if the original did, but since they are not implemented entirely in the bitcode, later interprocedural analysis of the lifted code is not possible.

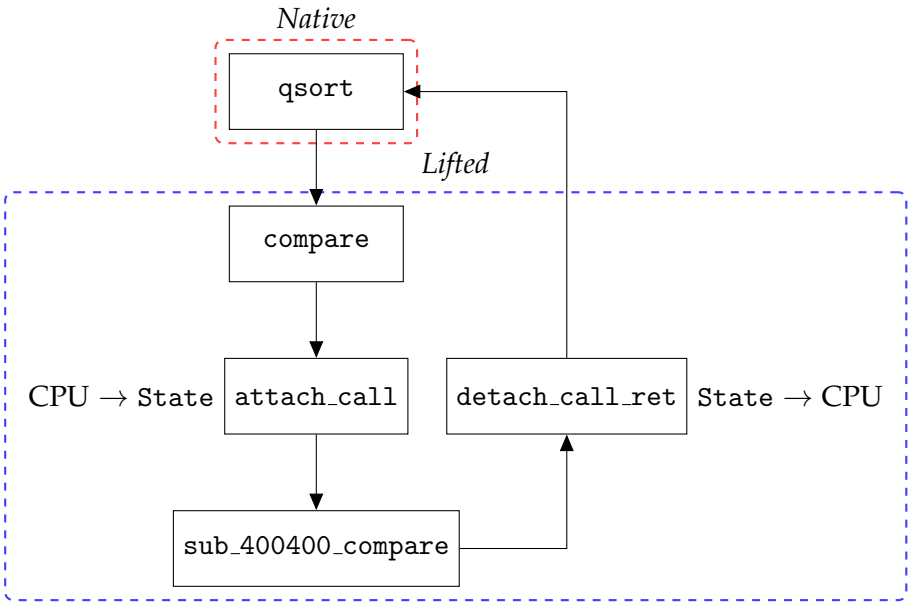

Figure 5.6: Synchronization via entrypoint

5.6.2 *Analysis mode*

The bitcode program that is supposed to be analyzed cannot contain or depend on subroutines written in assembly; therefore, no synchronization with native state can be performed – every part of the simulation must be done entirely in bitcode.

Since the assembly subroutine that would pass the `State` structure to the `main` function cannot be used, the `State` is defined as a global variable. Without the native state, the native stack pointer together with the stack itself are unavailable as well. However as the stack is essential to the simulation of the original program, it is defined as a global variable – a fixed-size array.

Calls to external functions are all reconstructed using the available function type (bitcode prototype is preferred over the default type created from information in the CFG file). In the case of functions without proper bitcode prototype, this method may not work. For example, `float` type of an argument will not be recovered properly, as each argument is of type `i64` by default and values of these types are typically passed in different registers.

The indirection wrappers in this mode have the argument count equal to a fixed number specified during the lift; types of the arguments are `i64` for the same reasons as mentioned in the Section 5.5.3. Implementation of the wrappers is straightforward, each argument is stored to its

corresponding register in the `State` structure based on the ABI and the function is called. After the function returns, the value from the register specified to hold the return value of type `i64` is loaded and the wrapper returns it.

Indirect function calls are handled in a way similar to the recompilation mode – the dispatch function (implementation of `__remill_function_call` intrinsic) casts its argument to the type of the indirection wrapper and calls it. Since the targets may be both external and internal functions, the dispatch function re-creates the call. The callee is casted to the type of indirection wrapper, proper arguments are loaded from the `State` structure and passed to the callee.

Indirect calls and indirection wrappers suffer from the same shortcomings as the re-creation of a call to the external without the bitcode type, i.e. if the original function had argument of a type that is not passed in the same registers as `i64` values then the called function will not behave as expected, since some of the arguments may be missing.

6 DYNINST FRONTEND & EVALUATION

This chapter is split into two parts. Section 6.1 outlines the basics of implementation of the frontend. Section 6.2 evaluates performance of the Dyninst frontend on the artificial test cases and compares all currently available frontends – Dyninst, IDA Pro [Hex19] and Binary Ninja [Vec19] on binaries from the GNU project [GNU19].

6.1 IMPLEMENTATION

`DyninstAPI` is an open source project which provides tools and libraries for binary instrumentation, analysis and modification [Par19]. In the implementation of the frontend only a subset of available libraries is used:

- `SymtabAPI` parses the symbol tables, object file headers and debug information. Both ELF and PE formats of binaries are supported.
- `ParseAPI` provides the user with control-flow oriented view of a program. By default it supports the same formats of binaries that are supported by the `SymtabAPI`.
- `InstructionAPI` is capable of decoding raw binary instructions.

The frontend does not use the modification capabilities of Dyninst – it is only used as a parsing tool to decode the information in the binary. The current implementation supports only 64-bit ELF binaries that were originally C programs. However, since Dyninst also supports the 32-bit ELF and PE binaries, there should be no fundamental obstacles to expanding the tool.

The goal of the frontend is to produce a CFG file (see Section 5.2) which describes the input binary. The fronted is implemented using several passes over the binary – one pass for each attribute of the Module entry in the CFG file. First pass is over externals, second pass is over sections of the binary and third pass is over internal functions.

6.1.1 *Disassemble Context*

As discussed in Section 5.2.1, recognizing all references is important for the success of the following lift; if some references are not recognized or

are recognized falsely, the resulting bitcode will not behave as the original binary did. The frontend tries to implement recognition of references on its own, with custom heuristics.

During the execution, the frontend keeps track of all objects and their addresses, forming a *disassembly context*. To decide whether an operand of instruction or an address in section contains a reference, the frontend uses a simple heuristic:

```
// address: target of the possible reference
// binary: object representing the original input binary
enum RefType Context::decide( uint64_t address, Binary binary ) {

  if ( this->contains( address ) )
    return RefType::Resolved;

  if ( !binary.is_in_some_section( address ) )
    return RefType::Never;

  if ( binary.is_in_code_section( address ) )
    return RefType::Maybe;

  if ( binary.is_in_data( address ) ||
       binary.is_in_rodata( address ) ||
       binary.is_in_bss( address ) ) )
    return RefType::Resolved;
  return RefType::Unknown;
}
```

The function `decide` tries four different condition to resolve a possible reference:

1. Tries to match the address with an object in the *context*.
2. If the target address is not contained in any section of the binary it cannot be a reference.
3. The address belongs to the `.text` section: If the binary contains a symbol table (is not stripped) then this situation should not occur often, as each function should be already present in the *context*. In case the binary is stripped, it is possible that `ParseAPI` heuristics missed a function that may start at this address; therefore, the frontend decides it is a reference and notifies caller that the target may be an unrecognized function.
4. The address is in one of the following sections: `.data`, `.rodata` or `.bss`: This case is really problematic as there are no easy heuris-

tics to be applied; the frontend always favors a reference which performed good in practice.

6.1.2 *Externals*

Since a list of all external symbols is included in the binary, `SymtabAPI` can be used to retrieve it. All information the CFG file needs about the external variables can be obtained using calls to the Dyninst libraries.

For external functions, a relocation address must be computed first (see Section 5.2.5). New base address outside of the address range of the binary is chosen (otherwise there may be conflicts with original objects) and each external function is mapped to a unique positive offset from this base address. Exact addresses or contiguity are not important – it is enough that each external function gets mapped to a unique address.

6.1.3 *Sections*

Each section is scanned for references using a naive default algorithm which traverses the section and reads an address every quadword (a pointer size in 64-bit ELF). The value stored at the read address is then resolved using the `decide` function.

Sections `.data` and `.rodata` contain most of the references originating from the original program. References in other sections are (mainly) for the linker or the system; for this reason these two sections are scanned for references more precisely. Addresses are read every 32-bit instead of 64-bit: jump tables, and sometimes strings, can start at an address aligned to a 32-bit boundary. However this is still a heuristic: if the original code contained packed structured, some references may not be recognized.

6.1.4 *Internal Functions*

A list of internal functions is retrieved from the symbol table if present; otherwise, heuristics offered by `ParseAPI` are used.

Possible functions can be also discovered in the `decide` function. If the target of a reference satisfies the third condition, it is a possible function. These possible entries are sorted into two sets: first contains addresses that are targets of references from data sections (*section_xrefs*), the second contains addresess recovered as targets of references in instructions (*inst_xrefs*). The *section_xrefs* set does not grow as functions are processed,

since all data sections have already been processed, while *inst_xrefs* starts empty.

Functions are procesed in the following order, adding all new possible addresses of functions to *inst_xrefs*:

1. All functions found by heuristics of the `ParseAPI`.
2. Functions starting at addresses from the *section_xrefs* set.
3. Functions starting at addresses from the *inst_xrefs* set, until the set is empty.

Unresolved references in *section_xrefs* can be entries of a jump table. *ParseAPI* provides a list of successors for every basic block of a function. If the number of successors is higher than three, the frontend verifies if the targets originate from a jump table – if all successors are in the *section_xrefs* set, they are removed, since they most likely do not represent the address of a new function but are instead jump table entries.

To find references in instructions, the frontend tries to evaluate the operands of each instruction and if the value is constant, it uses the `decide` heuristic to determine presence of a reference. In a position independent binary, immediate values are never references, since if an operand of instruction is an address, offset to the instruction pointer is always used.

6.2 EVALUATION

To run the tests McSema was compiled with LLVM version 4.0 on a machine with operating system Ubuntu 18.04. Since there is no clearly defined equivalence between the CFG files (there are programs for which no frontend can create a correct CFG file), the files are not compared directly. Instead the CFG file is lifted and the bitcode is recompiled, resulting in a recompiled binary that can be tested against the original.

6.2.1 *Artificial programs*

First set of test programs is made of artificial programs. Each program from the set demonstrates only simple functionality; a program with one simple `for` loop or a program calling one external function are examples of such artificial programs. The functionality corresponds to standard or commonly used constructs in the C language. There are some C++ programs in the set which tests classes (methods and attributes), inheritance and global variables. The Dyninst frontend can successfully decompile

Test case	Number of tests	Binary Ninja	Dyninst	IDA Pro
`awk`	2	Error	Error	2
`bash`	2	Error	Error	Error
`cat`	4	Error	4	4
`echo`	3	0	3	3
`grep`	5	Error	2	4
`gzip`	6	0	5	4
`ld`	2	0	0	1
`ls`	5	0	2	5
`perl`	2	Error	0	Error
`readelf`	7	Error	5	6
`sed`	5	0/5	2	5
`xz`	4	Error	4	4

Table 6.1: Evaluation of all frontends, binaries from repositories. Error means that the recompiled binary could not be produced.

each test program from the set. For the complete list and sources see Appendix B.

6.2.2 *Real-life binaries*

In this subsection binaries from the GNU project [GNU19] are tested. To put performance of the Dyninst frontend into perspective, the results of the same tests are shown for other two frontends as well.

Each program is lifted once and then run with several different inputs. One of the tests is always a simple `--help` as the only command line argument. For all technical details see Appendix B. There are two types of results for each program:

- **Error**: recompiled binary to be tested could not be produced. There are several possible reasons: the corresponding CFG file could not be produced, the lift process did not end successfully or the recompilation failed (errors or timeout).

- **Successful/total**: number of tests that were successful slash number of tests.

Table 6.1 shows results for binaries from Ubuntu repositories. These are all stripped and position independent. The Binary Ninja frontend performs the worst – every test case is either an *Error* or no successful test.

Test case	Number of tests	Binary Ninja	Dyninst	IDA Pro
`cat`	4	0	4	4
`echo`	3	0	3	3
`gzip`	6	Error	5	5
`ls`	5	Error	5	5
`readelf`	7	2	5	6
`du`	5	0	5	5
`objdump`	6	Error	0	0
`sha256sum`	5	0	5	5

Table 6.2: Evaluation of all frontends, manually compiled binaries. Error means that the recompiled binary could not be produced.

The IDA Pro frontend performed the best, it had the most successful tests for given binary with exception of `gzip`. The Dyninst frontend had significantly worse results on three of the total twelve binaries.

Only `perl` and `bash` were failures using all frontends; in every other case, at least one of them was able to successfully complete some tests.

Table 6.2 shows results for manually compiled binaries. These binaries are position-dependent (compiled with the `-nopie` flag) and contain symbol tables; with the exception of `gzip` which is position-dependent and stripped.

The only binary that no frontend was able to lift at least with partial success is `objdump`. Both the IDA Pro frontend and the Dyninst frontend produced a recompiled binary which did not pass any tests. Like with to previous results, the Binary Ninja frontend performs the worst, while the Dyninst frontend and the IDA Pro frontend peformance is almost the same.

7 CONCLUSION

This thesis presented the overview of several selected topics related to the decompilation. The inner workings of the McSema were described, therefore the thesis can serve as a future reference for developers who consider to use the tool.

In this thesis a new frontend for McSema was implemented, based on the open-source project Dyninst. It allows McSema to be used as fully open-source project; no mandatory component is locked behind proprietary license. Its performance was compared to the performance of the other frontends which used the proprietary tools, on programs that are used daily on computers with distributions of UNIX operating system. The Dyninst frontend performed better than the Binary Ninja frontend and only slightly worse than the IDA Pro frontend – there was significant difference only with three tested binaries (totally twenty was tested).

7.1 FUTURE WORK

While the frontend implemented in this thesis allows McSema to be used without proprietary software, there are still many options to improve it. Support of other file formats that are supported by Dyninst (PE or 32-bit ELF) would increase the spectrum of binaries the frontend can handle.

Currently, the frontend can work only with binaries that were originally written in the C programming language and while some specifics of the C++ language are supported, the exceptions are not. Successfully parsing the exception information from the binaries would allow the frontend to work on previously unavailable targets.

APPENDIX

A

BUILD PROCESS

The archive `mcsema.tar.gz` contains all th McSema sources needed to compile the program. Both the Dyninst sources and the `cxx-common` sources are included as well for convenience.

```
$ tar xvf sources.tar.gz # Extract the content of the archive
```

If the target operating system is Ubuntu, the prebuilt libraries of dependencies are available; to build the program simply use the `build.sh` script:

```
$ cd remill
$ scripts/build.sh
```

To compile on different operating system than Ubuntu, the dependencies must be compiled manually – refer to the McSema's `README` for more information.

To build the Dyninst frontend, Dyninst itself must be built first (version 9.3.x or version 9.3.2) and the cmake command `find_package` must be able to find the installation (set the `CMAKE_PREFIX_PATH` appropriately). The path to installed libraries may be added to `LD_LIBRARY_PATH` and as well. After Dyninst is compiled, the frontend can be built with a special option passed to the build script:

```
$ cd remill
$ scripts/build.sh --dyninst-frontend
```

After the build ends with success, the `remill-build` directory is created.

```
$ cd remill/remill-build
$ cd tools/mcsema
$ ./mcsema-lift-4.0 --help # mcsema-lift binary
$ ls mcsema/Arch/X86/Runtime/libmcsema_rt64-4.0.a # runtime libraries
$ cd tools/mcsema_disass/dyninst/
$ ./mcsema-dyninst-disass --help # the Dyninst frontend executable
```

Each of the directories mentioned bellow has its own `README.md` with additional information:

```
$ cd remill # sources of Remill library
$ cat README.md
$ cd tools/mcsema # sources of Mcsema
$ cat README.md
$ cd tools/mcsema_disass/dyninst # sources of the Dyninst frontend
$ cat README.md
```

B

TESTS

The archive `tests.tar.xz` contains all the test data and scripts used in evaluation.

```
$ tar xvf tests.tar.xz # Extract the content of the archive
```

There are six directories, two for each frontend corresponding to data in Table 6.2 and Table 6.1. Each contains following files and directories:

- `run.sh`: helpful wrapper that calls the main test script.
- `run_tests.py`: main evaluation script. Requires Python of version 2.7.
- `bin`: directory that contains the original binaries.
- `cfg`: in case of the frontends using proprietary disassemblers this directory already contains CFG files.
- `lift_program.py`: script used to create CFG files.
- `libc`: directory that contains bitcode prototypes of standard C functions.
- `Makefile`: used to create CFG files, invokes `lift_program.py` and copies the results into appropriate locations.

To reproduce the results for the Dyninst frontend:

```
$ make all
$ ./run_tests.py --help # Will print required flags
```

For other frontends `make` command should be skipped, as the CFG files are already generated. During testing, for each test case separate test directory is created. Names of the test case directories start with prefix `build_`, followed by the name of the tested program. It is recommended to remove the test directories before invoking the test script again.

B.1 ARTIFICIAL PROGRAMS

Programs evaluated in Section 6.2.1 are stored in the directory with the Dyninst frontend sources:

```
$ cd remill/tools/mcsema/tools/mcsema_disass/dyninst/tests
```

The directory also contains the script used to run the tests. The test script requires Python of version 3.5 or higher. Compilers used to compile the sources into binaries to be tested were `clang` and `clang++` both version 4.0.

B.2 TEST NOTES

The IDA Pro frontend reached timeout when recompiling `perl` binary. Reason is the size of the generated CFG file.
The IDA Pro frontend reached timeout with one test: `test_grep_i`.

BIBLIOGRAPHY

[Amda] *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*. 24592. Version 3.22. URL: https://www.amd.com/system/files/TechDocs/24594.pdf (visited on 05/16/2019).

[Amdb] *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. 24594. Version 3.26. May 2018. URL: https://www.amd.com/system/files/TechDocs/24594.pdf (visited on 05/16/2019).

[Aho+06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[BR07] G. Balakrishnan and T. Reps. "DIVINE: Discovering Variables in Executables". In: *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'07. Nice, France: Springer-Verlag, 2007, pp. 1–28. ISBN: 978-3-540-69735-0. URL: http://dl.acm.org/citation.cfm?id=1763048.1763050.

[Cyt+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925.

[Din+19] A. Dinaburg, A. Kumar, P. Goodman, A. Gario, and G. Reece. *McSema*. 2019. URL: https://www.trailofbits.com/research-and-development/mcsema/ (visited on 05/15/2019).

[EW04] M. V. Emmerik and T. Waddington. "Using a decompiler for real-world source recovery". In: *11th Working Conference on Reverse Engineering*. 2004, pp. 27–36. DOI: 10.1109/WCRE.2004.42.

[GNU19] GNU Project. *The GNU Operating System*. 2019. URL: https://www.gnu.org/ (visited on 05/15/2019).

[Hex19] Hex-Rays. *IDA Pro disassembler and debuger*. 2019. URL: https://www.hex-rays.com/products/ida/ (visited on 05/15/2019).

[Int] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 325462-069US. Jan. 2019. (Visited on 05/16/2019).

[Jan14] Jan Hubička and Andreas Jaeger and Mark Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*. Version 0.95. Jan. 2014. URL: http://refspecs.linux-foundation.org/elf/x86_64-abi-0.95.pdf (visited on 05/16/2019).

[LLV19] LLVM Project. *LLVM Language Reference Manual*. 2019. URL: http://llvm.org/docs/LangRef.html (visited on 05/15/2019).

[Lat19] C. Lattner. *The LLVM Compiler Infrastructure Project*. 2019. URL: http://llvm.org/ (visited on 05/15/2019).

[LAB11] J. Lee, T. Avgerinos, and D. Brumley. "TIE: Principled Reverse Engineering of Types in Binary Programs". In: *Network and Distributed System Security Symposium (NDSS) 2011*. Jan. 2011. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/lee.pdf.

[Lev99] J. R. Levine. *Linkers and Loaders*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1558604960.

[MY60] R. McNaughton and H. Yamada. "Regular Expressions and State Graphs for Automata". In: *IRE Transactions on Electronic Computers* EC-9.1 (1960), pp. 39–47. ISSN: 0367-9950. DOI: 10.1109/TEC.1960.5221603.

[Par19] Paradyn Project. *Dyninst: Putting the Performance in High Performance Computing*. 2019. URL: https://dyninst.org/ (visited on 05/15/2019).

[Roč15] P. Ročkai. "Model Checking Software [online]". Doctoral theses, Dissertations. Masaryk University, Faculty of Informatics, Brno, 2015. URL: http://is.muni.cz/th/139761/fi_d/.

[TIS95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. May 1995. URL: https://refspecs.linuxfoundation.org/elf/elf.pdf (visited on 05/16/2019).

[The17] The NASM Development Team. *NASM – The Netwide Assembler*. Version 2.14.02. 2017. URL: https://www.nasm.us/xdoc/2.14.02/nasmdoc.pdf (visited on 05/19/2019).

[TDC10] K. Troshina, Y. Derevenets, and A. Chernov. "Reconstruction of Composite Types for Decompilation". In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 2010, pp. 179–188. DOI: 10.1109/SCAM.2010.24.

[TF01] Tzi-Cker Chiueh and Fu-Hau Hsu. "RAD: a compile-time solution to buffer overflow attacks". In: *Proceedings 21st International Conference on Distributed Computing Systems*. 2001, pp. 409–417. DOI: 10.1109/ICDSC.2001.918971.

[Vec19] Vector 35. *Binary Ninja: A new kind of reversing platform*. 2019. URL: https://binary.ninja/ (visited on 05/15/2019).

[Wir96] N. Wirth. "Extended backus-naur form (ebnf)". In: *Iso/Iec* 14977 (1996), p. 2996.

[bin] binutils. *Using as*. Version 2.32. URL: https://sourceware.org/binutils/docs/as/index.html (visited on 05/19/2019).