Decompilation of Binaries into LLVM IR for Automated Analysis

by

Tejvinder Singh Toor

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Complexity in malicious software is increasing to avoid detection and mitigation. As such, there is greater interest in using automation for reverse engineering. Current state-of-the-art tools use proprietary intermediate representations (IR) in decompilation and lack open-source development. LLVM IR has emerged as a candidate for a reverse engineering IR as it is already a mature tool for compilation and has a wide set of existing analysis tools. In 2019, the NSA released the Ghidra reverse engineering framework as a free and open-source alternative. In this thesis, we examine the development and application of IRs in Ghidra for lifting to LLVM IR and evaluating the efficacy of that lifting. Of interest was lifting at both the disassembly and decompilation stages of Ghidra. We developed two tools: Ghidra-to-LLVM and Ghidrall. The former uses Ghidra's Low P-Code IR for a disassembling lifter while the latter uses Ghidra's decompilation data structures as a decompiling lifter. Lastly, we test the efficacy of Ghidrall as an input for automated solving and against another lifter. Our results show that Ghidra is effective and has promise as an input for future LLVM-based reverse engineering technologies.

Acknowledgements

I would like to thank my advisor, Arie Gurfinkel for his mentorship and support.

I would like to thank my readers Mahesh Tripunitara and Werner Dietl.

I would like to thank my colleagues Hung, Thibaud, and Yitong for their support.

I would also like to thank my friends and family for their support as well.

Dedication

This is dedicated to the ones I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software reverse engineering is a useful technique for finding vulnerabilities in black-box testing environments as it mimics the same methods a malicious actor would use. It is also becoming an increasingly difficult challenge[1] due to growth in software complexity. Stuxnet[2], for example, is a widely considered an extremely complex malware and over ten years old. In order to evade defenses and mitigation, an arms race has resulted in complex malware that are not easy to reason about. Research has become more expensive due to the fundamental requirement of human analyst time and experience. As such, between reversing malicious programs as well as defensive testing of binary programs, there has been growing interest in improving workflows and reverse engineering automation. Current research workflows involve tools like IDA Pro from Hex-Rays[3]. IDA Pro includes a disassembler, a decompiler, as well as debugging and dynamic analysis tools. The primary downsides for IDA Pro are learning curve, ease-of-access, and lack of open-source. Previously, price was a major factor but pressure from competitors has forced Hex-Rays to reduce the cost.

Automation is already a hot topic in reverse engineering. In 2016, DARPA ran the Cyber Grand Challenge[4]. The goal of the challenge was to produce fully-automated software that would be able to discover and patch novel vulnerable binaries. From this competition a number of tools were produced, some of which are being adapted to commercial products like Mayhem[5], Xandra[6], and Shellphish[7]. Additionally, other assisted-automated tools for reverse engineering exist like Angr[8]. These tools require human guidance to work but speed up the process. Another automated method of vulnerability discovery is fuzzing, where programs like American Fuzzy Lop (AFL)[9] have discovered vulnerabilities in hundreds of programs.

Existing state-of-the-art reverse engineering tools lack standards for open source work and have implementations of the same functionality. Most of these tools run on their own intermediate representations for decompiling and analysis and are not transferable between one another. LLVM IR[10] already exists as a standard for compilation. It is mature and well-tested, and has a mature set of tools that can be used for program analysis like KLEE[11] and SeaHorn[12]. However, unlike for compilation there is no standard process for decompilation. Disassembling and decompilation of binaries to LLVM IR is known as *lifting*. We distinguish between the two with the terms *disassembling* lifter and *decompiling* lifter. There are currently a few LLVM lifters like McSema[13], McToll[14], and RetDec[15].

In 2019, the National Security Agency released its own free and open-source tool, Ghidra[16]. It includes a disassembler, a decompiler, a plugin interface, and a debugger. Of interest are its intermediate stages. *Low-P-Code*[17] is the IR that architectures are translated to before decompilation is performed. *High P-Code* introduces static single-assignment operations and markers for further decompilation phases. Through modifying the decompiler, we expose a third intermediate representation, which we call *Decompilation Data Structures*. This IR exposes P-Code and other decompilation information before it is translated to the C-like pseudo-code that is presented to the user in the Ghidra UI. The challenges for this work were exposing the different layers of Ghidra to develop tools, translating instructions between Ghidra and our tools, and managing machine emulation at different levels.

The contributions of this thesis as follows:

- The design of a disassembling lifter, **Ghidra-to-LLVM**, based on Ghidra's Low P-Code. This tool was developed to a proof of concept level.
- The design of decompiling lifter, **Ghidrall**, based on the Ghidra decompiler's internal data structures. Ghidrall was the primary effort of this thesis.
- An evaluation of Ghidrall against McSema using instrument test programs with the SeaHorn verification framework. We find a 15% improvement in accuracy in preserving program functionality with Ghidrall. Ghidra-to-LLVM was not developed to the same standard as Ghidrall; as such it was not evaluated in the same testing scheme as McSema and Ghidrall.

Chapter 2

Background

In this chapter we present information regarding LLVM IR and P-Code for this thesis.

2.1 LLVM

The LLVM Project is a collection free, modular, and open compiler-related technologies. The ecosystem is designed to interface with new programming languages and machine architectures through the compiler frond-end and back-end, respectively. This feature is enabled through its intermediate representation, LLVM IR[10]. LLVM IR is a strongly typed and single-static assigned (SSA). LLVM IR is the output used by lifters in reverse engineering processes.

Concept	**LLVM Example**
Function Declaration	declare {i32, i1} @add_with_overflow(i32 %1, i32 %2)
Global Variable	@X = internal global i32 0
Control Flow	br label %5
GEP Instruction	%155 = getelementptr i8, i8* %5, i64 0

Table 2.1.1: Examples of LLVM IR

Table 2.1.1 illustrates a few examples of LLVM IR. The readable representation of LLVM IR is emitted as a `.ll` file. Each file consists of a module, which corresponds to the input programs as a translation unit. Multiple module files may be linked by the LLVM linker. Each module consists of functions, global variables and symbol table entries. There are two types of identifiers in LLVM: the global identifier (`@`) and the local identifier (`%`). Line 1 of the table illustrates an example of the global identifier used in a function declaration, while line 4 is an example of the local modifier used to identify an LLVM *register*. Registers in LLVM IR refer to single-use variables in SSA.

Functions in LLVM can either be declared or defined. Declared functions are used as placeholders until linking defines them. Each function takes in a series of inputs on the right side and emits a single output. The type of the output is determined by either instruction type or the input values. For instance, integer addition with the `add` instruction must take two inputs of the same size and emits an output of that same size. Each function is made up of one or more basic blocks in a Control-Flow Graph (CFG). Each block consists of a label, instructions, and a terminator instruction. Line 3 of the table illustrates an example of a terminator instruction.

Values in LLVM can either be LLVM registers (defined with SSA), constants, or globals. All values in LLVM are bit-arrays (written as `i32` for a 32-bit integer). Pointers in LLVM are defined with an additional `*` affix. Line 4 is an example of the LLVM `GEP` instruction, which is used for accessing values in structures, pointer arithmetic, and dereferencing.

2.2 Ghidra

Reverse engineering tools are typically packaged into a framework. The framework allows an analyst to develop a simple workflow. Ghidra, a reverse engineering platform, was developed by the National Security Agency and released to the general public as a free and open-source project in 2019. It consists of a disassembler and decompiler, as well as a suite of visualization and editing tools. Ghidra also includes a plugin interface so users can access the API. Ghidra goes through a similar transformation process as LLVM compilation, where there are a series of stages for decompilation and different front-ends (called *processors*) and back-ends for decompiled code.

Figure 2.2.1 illustrates the data-flow in the Ghidra decompilation process and the different intermediate representations that can be accessed at each stage. The first stage is where raw P-Code is generated. We refer to this format as *Low P-Code*[17]. This P-Code is generated by processors that are unique to specific system architectures. Ghidra-to-LLVM uses this output as a source for lifting a binary to LLVM. The second type of IR is accessible after the CFG Recovery and Annotation phase of Ghidra. This process recovers some control flow and introduces markers for further decompilation stages. No tool was developed for *High P-Code* as the markers are not directly translatable to LLVM IR. Two further stages

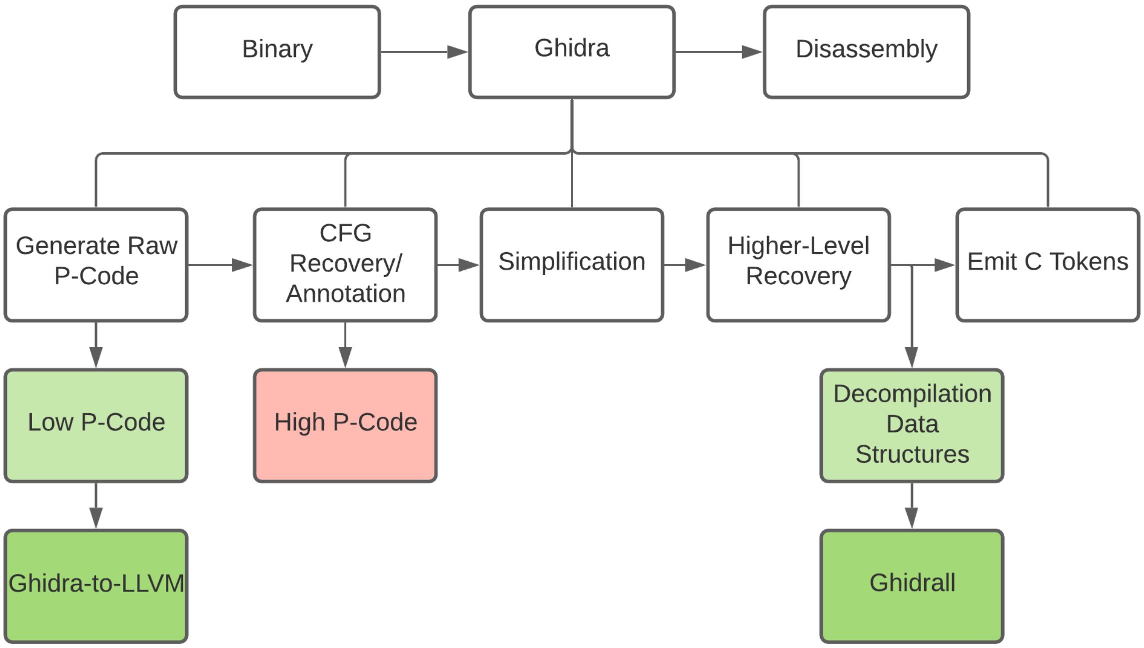

Figure 2.2.1: Internal Representations of Ghidra Data Flow

are applied to decompile the P-Code before translating it to a programming language for human-readability. Ghidra defaults to a pseudo-C type of representation, but it is possible to modify the Ghidra source to access P-Code before it is translated to pseudo-C. We define this stage of the internal representation as *Decompilation Data Structures* as it consists of complex data-structures that contain P-Code as well as other information about memory and control-flow. This intermediate representation is consumed by Ghidrall to produce LLVM IR.

2.2.1 Low P-Code

Figure 2.2.2 is a snippet of a few P-Code instructions. Low P-Code retains references to architecture specific values. In lines 2 and 3 we see references to the carry flag (`CF`) and the zero flag (`ZF`), which are in this case registers specific to x86 assembly. P-Code instructions follow polish notation and generally require their output size to match their input sizes. Input and output values are referred to as *varnodes*. Varnodes in P-Code consist of an

```
1 $U34b0:1 = INT_SLESS $Ub7d0, 0:8
2 $U3450:1 = INT_AND $U3440, CF
3 CBRANCH A_00100532:8, ZF
```

Figure 2.2.2: Snippet of Low P-Code

address space, an offset into that space and a size. The segment after the colon of a value is the size of the variable in bytes.

Address Spaces

Address spaces are generalizations for memory in Ghidra. It is a sequence of bytes that can be written to and read from. Each byte has an address associated with it. There are few types of address spaces:

- `ram` space is used to model the RAM on a real processor. The value `A_00100532:8` in line 3 is an example of an address in `ram` space.
- `register` space is used to define architecture-specific registers like `EAX` or `CF`.
- `constant` space is used to define constant values that are accessed by instructions. `0:8` in line 1 is an example.
- `temporary` or `unique` space is used for temporary values. Varnodes like `$U3440` are temporary values.

2.2.2 High P-Code

High P-Code is accessible and includes some control-flow graph recovery as well as annotations for further decompilation. Figure 2.2.1 shows the new P-Code operations that are defined in High P-Code. All of these operations can be translated to LLVM except for the `INDIRECT` operation. This does not have any explicit meaning and is used to mark varnodes as being potentially implicitly modified by another instruction.

High P-Code	**Explanation**
MULTIEQUAL	Phi node for SSA
INDIRECT	Marker for decompiler for indirect change
PTRADD	Pointer addition
PTRSUB	Pointer subtraction
CAST	Type casting

Table 2.2.1: P-Code Operations Introduced at High P-Code Stage

2.2.3 Internal Decompilation Data Structures

The internal decompilation data structures are accessible by modifying the Ghidra decompiler. P-Code instructions can be accessed by iterating over functions and emitting their structures in blocks. The instructions are the same as High P-Code but have `INDIRECT` operations removed. Additional information like function paramaters, function variables, and stack information can be found in these data structures.

2.3 Translating P-Code to LLVM IR

Figure 2.3.1 shows the translation between machine instructions and Low P-Code. Each machine instruction maps to one or more P-Code operations. In this case, all of the steps involved in translating `MOV EAX,dword ptr [ESP + local_14]` to P-Code are broken down and shown in red. Each of these operations is then mapped to a P-Code operation, which is shown in green.

Figure 2.3.2 maps each of the previous P-Code operations to LLVM IR. Much like with the machine code, each P-Code operation maps to one or more LLVM IR lines, shown in white. Both LLVM IR and P-Code integer operations require that both inputs and the output be the same size and type. P-Code does not distinguish between pointer and integer types like LLVM IR does, so additional processing needs to be added during lifting.

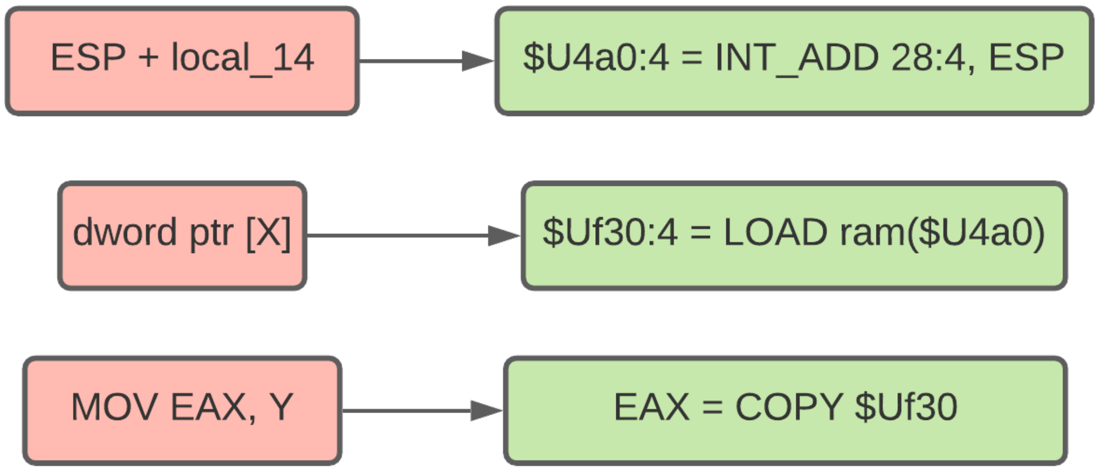

Figure 2.3.1: Translation of x86 Assembly to Low P-Code

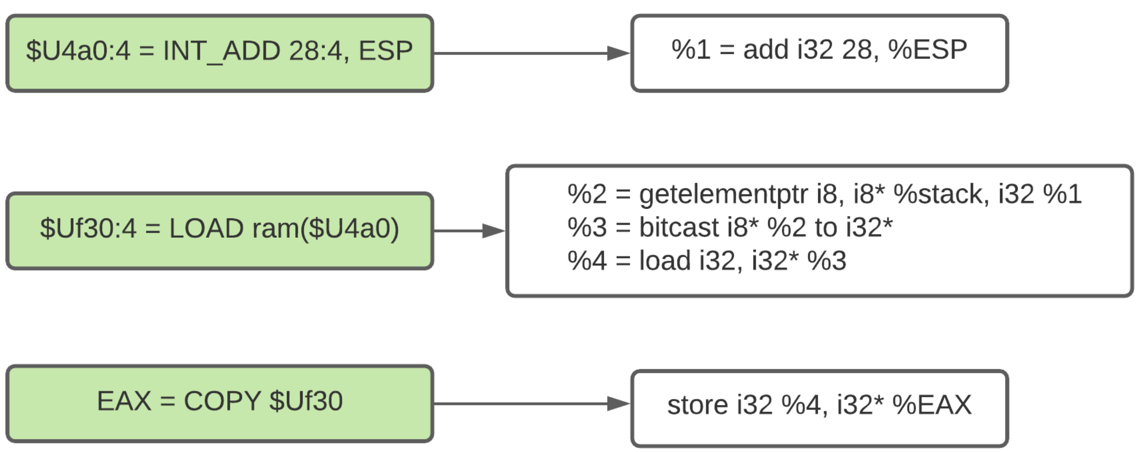

Figure 2.3.2: Translation of Low P-Code to LLVM IR

Chapter 3

Ghidra-to-LLVM

3.1 Overview

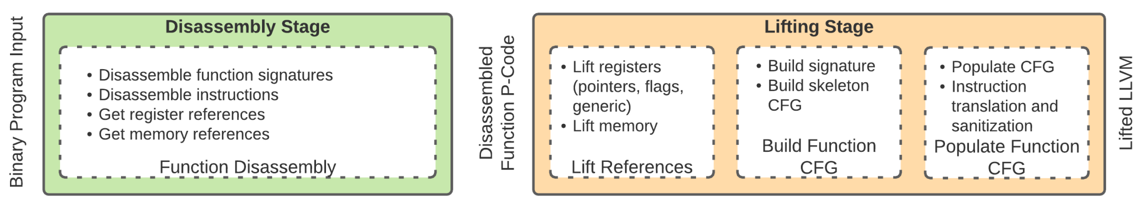

Figure 3.1.1: Overview for Ghidra-to-LLVM.

Figure 3.1.1 shows an overview of the Ghidra-to-LLVM program flow. Overall, the program takes in a compiled binary, disassembles it into intermediate decompilation data structures, and then lifts the structures into valid LLVM IR. In this chapter the disassembly and lifting stages are presented in detail, as well as an example of a buffer overflow vulnerability being preserved after lifting.

In the *disassembly* stage Ghidra-to-LLVM needs to recover the function signature, disassemble instructions, and maintain references to registers, memory, and the stack. These features form part of the main challenge with Ghidra-to-LLVM — since it is so low level, there is a need to emulate the machine to maintain the logic of the program without decompiling the program. This creates a machine model that is specific to this tool and platform agnostic.

In the *lifting* stage there are four steps. First, references to memory and registers are defined in LLVM as global variables. Then, each function has its function signature defined and its CFG skeleton built. This CFG is then populated with LLVM instructions derived from the Low P-Code instructions. Finally, the entire output is verified as valid LLVM before being outputted.

The source for Ghidra-to-LLVM and its test can be found at the following webpage: https://github.com/toor-de-force/Ghidra-to-LLVM.

3.2 Disassembly

Algorithm 1: Ghidra-to-LLVM Disassembly Algorithm

Result: Low P-Code XML

foreach *function* **do**
  $DisassembleFunctionSignature(function)$;
  foreach *instruction* **do**
    $DisassembleInstruction(instruction)$;
  end
end
$EmitRegisterMemoryReferences$;

The disassembly stage of Ghidra-to-LLVM is entirely self-contained within a headless plugin for Ghidra. Algorithm 1 illustrates the top-level steps it takes to disassemble the program and emit the intermediate low P-Code data structures. An input program is passed into the disassembler plugin using the `analyzeHeadless` utility provided by the Ghidra API.

Using the Ghidra plugin API, functions are collected and passed through the `DisassembleFunctionSignature` procedure, which recovers function return types, paramaters, address, and name. The function's constituent assembly instructions are then iterated over and passed through the `DisassembleInstruction` procedure, which recovers each assembly instruction's address, constituent P-Code operations and their input and output values. Finally in the global scope the `EmitRegisterMemoryReferences` procedure register and memory references are collected along with their sizes and addresses to facilitate lifting later on.

3.2.1 Disassemble Function Signature

Algorithm 2: *DisassembleFunctionSignature*

Result: Function Low P-Code XML
Emit *function* name;
Emit *function* address;
Emit *function* output type;
foreach *input* **do**
    Emit *input* name;
    Emit *input* type;
end

The `DisassembleFunctionSignature` procedure is outlined in Algorithm 2. It takes in a function and emits the name, address, return type as well as the input names and types. The name and address are used to maintain references to the function in lifting since it is legal in P-Code to refer to a function by either when performing calls. Output type is assumed to be void if it is impossible to confidently recover the function type. Figure 3.2.1 is an example comparing the source of `func0` and the disassembly XML. In this example the disassembly is fully accurate.

3.2.2 Disassemble Instruction

Algorithm 3: *DisassembleInstruction*

Result: Instruction Low P-Code XML
Emit *assembly* address;
foreach *P-Code op* **do**
    Emit *P-Code op* output size and storage type;
    foreach *input* **do**
        Emit *input* size and storage type;
    end
    Emit *P-Code op* name;
end

Algorithm 3 illustrates the steps required to disassemble a single instruction for the procedure `DisassembleInstruction`. Instruction disassembly in Ghidra-to-LLVM treats each

```c
void func0(int x) {

  int n=INT_RAND;
  if (n==4 && x < 10) {
    func1(n, x);
  }
}
```

(a) Source code

```xml
<function address="0010071c" name="func0">
  <output type="void"/>
  <input name="x" type="int"/>
  <instructions>
     ...
  </instructions>
</function>
```

(b) Ghidra-to-LLVM Intermediate XML Output

Figure 3.2.1: Comparison of source and intermediate XML Output

assembly instruction as its own block. As P-Code operations do not cover all possible operations in a single instruction set, one assembly instruction can map to one or more low P-Code operations. Additionally, implicit changes like flag settings need to be explicitly defined.

Of note is the *storage* field, which keeps track of the Ghidra storage type. The possible options are `register` (a register as defined by the instruction set), `memory` (a memory location), `constant` (a constant integer value), or `unique` (Ghidra's temporary type used for temporary values). Figure 3.2.2 is an example of the output of the disassembly stage for the assembly corresponding to branch if not equal.

3.2.3 Emit Register and Memory References

The final procedure in the disassembly stage is `EmitRegisterMemoryReferences`. As register and memory references are hit in the previous procedures, these values are kept track of and emitted as separate lists along with their sizes. This step is there to facilitate the lifting stage where these values must be defined in the global scope and not within

```
<instruction_0>
  <address>0010071c</address>
  <pcodes>
        <pcode_0>
          <output size="8" storage="unique">u_2510:8</output>
          <name>COPY</name>
          <input_0 size="8" storage="register">RBP</input_0>
        </pcode_0>
        <pcode_1>
          <output size="8" storage="register">RSP</output>
          <name>INT_SUB</name>
          <input_0 size="8" storage="register">RSP</input_0>
          <input_1 size="8" storage="constant">0x8</input_1>
        </pcode_1>
        <pcode_2>
          <name>STORE</name>
          <input_0 size="8" storage="constant">0x1b1</input_0>
          <input_1 size="8" storage="register">RSP</input_1>
          <input_2 size="8" storage="unique">u_2510:8</input_2>
        </pcode_2>
  </pcodes>
</instruction_0>
```

Figure 3.2.2: Ghidra-to-LLVM XML instruction output

functions in order to work with LLVM. Figure 3.2.3 shows an example of both outputs; the registers seen here are `x86` registers.

3.3 Lifting Stage

The lifting stage of Ghidra-to-LLVM builds the LLVM files and validates them. Algorithm 4 illustrates the top-level steps it takes to lift the program and output the final LLVM file. The XML output from the disassembly stage is used to perform the lifting.

First, `LiftRegistersandMemoryReferences` produces the LLVM variables that reference register and memory locations. These need to be performed separately as these belong to the global scope and not any function. Then each function is first built with a skeleton CFG

```
<globals>
  <register name="CF" size="1"/>
  <register name="RSP" size="8"/>
  <register name="OF" size="1"/>
  <register name="SF" size="1"/>
  <register name="ZF" size="1"/>
  <register name="RAX" size="8"/>
  <register name="RIP" size="8"/>
    ...
  <register name="RBX" size="8"/>
</globals>
<memory>
  <memory name="A_00300fe8:8" size="8"/>
  <memory name="A_00301010:1" size="1"/>
    ...
  <memory name="A_001007b0:8" size="8"/>
</memory>
```

Figure 3.2.3: Example of XML output for register and memory references

in `BuildFunctionCFG` before populating the function's CFG in `PopulateFunctionCFG`.

3.3.1 Lift Registers and Memory References

Algorithm 5 illustrates the steps required to lift the global scope registry and memory variables. Since register storage types in Ghidra do not necessarily include the correct sizing nor pointer types additional analysis is required. Register values are compared against known architecture-specific registers before lifting them to their respective types. If neither, the register is a regular register and no additional processing is needed. Memory locations are more straightforward and can be lifted without any additional processing. Figure 3.3.1 shows an example of XML inputs and their resulting LLVM outputs.

3.3.2 Build Function and CFG

Algorithm 6 shows the steps to construct the skeleton of an LLVM function. This needs to be a separate step to ensure references to branch locations and functions exist before processing instructions. Each assembly instruction address is treated as its own basic block.

Algorithm 4: Ghidra-to-LLVM Lifting Algorithm

```
Result: Lifted LLVM
LiftRegistersandMemoryReferences;
foreach function do
    BuildFunctionCFG(function);
    PopulateFunctionCFG(function);
end
```

Algorithm 5: *LiftRegistersMemory*

```
Result: Register and Memory Data Structures in LLVM
foreach register do
    if register is a flag then
        Lift flag register;
    else if register is a pointer then
        Lift pointer register;
    else
        Lift generic register;
    end
end
foreach memory do
    Lift memory location;
end
```

Algorithm 6: *BuildFunctionCFG*

```
Result: Function Structure in LLVM
foreach function do
    Build function type;
    Build entry block;
    foreach instruction address do
        Build instruction block;
    end
end
```

```
<globals>
  <register name="CF" size="1"/>
  <register name="RSP" size="8"/>
  <register name="OF" size="1"/>
  <register name="SF" size="1"/>
  <register name="ZF" size="1"/>
  <register name="RAX" size="8"/>
  <register name="RIP" size="8"/>
     ...
  <register name="RBX" size="8"/>
</globals>
<memory>
  <memory name="A_00300fe8:8" size="8"/>
  <memory name="A_00301010:1" size="1"/>
     ...
  <memory name="A_001007b0:8" size="8"/>
</memory>
```

(a) Intermediate XML

```
@"RSP" = internal global i8* null
@"RIP" = internal global i8* null
@"CF" = internal global i1 0
@"OF" = internal global i1 0
@"SF" = internal global i1 0
@"ZF" = internal global i1 0
@"RAX" = internal global i64 0
@"A_00101008:8" = internal global i64 0
@"A_00100000:8" = internal global i64 0
@"A_00101010:8" = internal global i64 0
```

(b) Final LLVM

Figure 3.3.1: Comparison of intermediate XML and LLVM for registers and memory

3.3.3 Populate Function and CFG

Algorithm 7 illustrates `PopulateFunctionStack` procedure. The population stage of the lifting process is where the vast majority of the lifting is done. A stack for each function is constructed to maintain the machine emulation requirements that are needed to analyze code at this level. Then for each P-Code operation we map it to an LLVM operation (`LiftOp`) and sanitize the inputs and outputs (`SanitizeOp`) as LLVM and P-Code do not follow the same rules for typing.

Ghidra-to-LLVM is able to recover control-flow graphs. The tool does this by treating each machine instruction as a single basic block, with its associated P-Code operations forming the instructions within it. Flow between blocks is either explicitly defined in the machine instructions, or is implicitly recovered as a fall-through to the subsequent block. Simplification passes are performed later on by LLVM optimization passes.

Algorithm 7: *PopulateFunctionCFG*

```
Result: Function Population in LLVM
foreach function do
    PopulateFunctionStack;
    foreach block do
        foreach op do
            LiftOp(op);
            SanitizeOp(op);
        end
    end
end
```

Function Stack

```
entry:
  %"stack" = alloca i8, i32 10485760
  %"stack_top" = getelementptr i8, i8* %"stack", i64 10485752
   store i8* %"stack_top", i8** @"RSP"
   br label %"0010066c"
```

Figure 3.3.2: Example of LLVM output of a function stack

Figure 3.3.2 is an example of how Ghidra-to-LLVM represents the stack. Each function in the program has its own stack, represented by a pointer to an arbitrarily large area of memory allocated using LLVM's `alloca` instruction. The top of the stack is then calculated and stored in the stack pointer `RSP`. As the stack pointer is a global register, when a function call is made a reference to the previous function's stack is maintained and arguments can be passed between functions. Function calls in Ghidra-to-LLVM are made without arguments. Arguments are instead passed through the stack.

Instruction Translation

No.	P-Code	LLVM IR via llvmlite
1	B = COPY A	%temp = load i8, i8* %A store i8 %temp, i8* %B
2	val:1 = LOAD ram(addr)	%val = load i8, i8* %addr
3	STORE ram(addr), val:4	store i32 %val, i32* %addr
4	BRANCH *[ram]addr	br label %"3"
5	CBRANCH *[ram]addr, val	br i1 %val, label %1, label %"2"
6	CALL *[ram]0x8048728b	call void @sym.path_start()
7	RETURN	ret void
8	out = INT_EQUAL A, 0:4	%out = icmp eq i32 %A, 0
10	out = INT_ADD A, 3:4	%out = add i32 %A, 3
11	out:1 = BOOL_XOR A, 1:1	%out = xor i1 %A, 1

Table 3.3.1: Mappings of P-Code to LLVM IR

Table 3.3.1 illustrates some examples of how operations are translated in `LiftOp`. Rows 1–3 are examples of how data P-Code operations are mapped to LLVM IR. Rows 4–7 illustrate examples of control flow instructions. Rows 8–11 are examples of arithmetic operations.

Instruction Sanitizing

P-Code and LLVM IR differ in how they treat operation inputs and outputs. For instance, LLVM IR typically requires inputs and outputs to be of the same size, while P-Code does not necessarily have the same requirement. Additionally, Ghidra operands do not explicitly keep track of whether or not the values they reference are pointers, while LLVM requires its registers to be explicitly defined as either pointers or non-pointers.

3.4 Example of Preservation of Buffer Overflow

In this section an analysis of a buffer overflow being preserved in lifting is presented. Appendix A includes the C source while the lifted `vuln.ll` can be found in Appendix B.

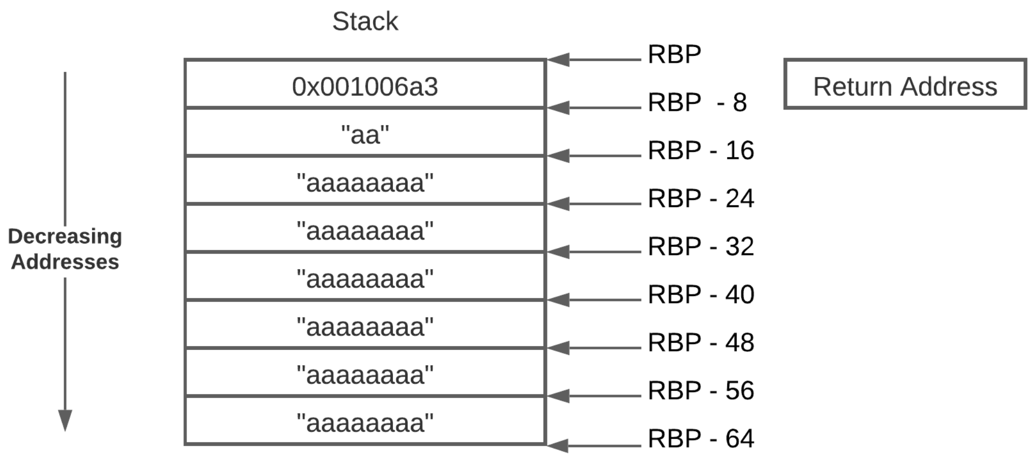

Figure 3.4.1: Call Stack Setup

Figures 3.4.1 and 3.4.2 show an example of a function call within Ghidra-to-LLVM. Ghidra prepares calls by placing the call arguments on the stack. Blocks `00100674` (lines 1–3) and `0010067e` (lines 4–6) form the beginning of the function call. In both of these instruction the string `"aaaaaaaa"` as in integer is stored in each of `RAX` and `RDX`. In the following blocks (`00100688` – `0010069c` these values are copied onto the stack at indexes off of the base pointer. For each of these the base pointer `RBP` is decremented and a portion of the input parameter is stored. In this case, the value `"aaaaaaaa"` was previously stored in the general purpose registers `RAX` and `RDX`. Each of these blocks was originally a `COPY` instruction, so it is represented as a load/store pair. The index register `RDI` is then used to store a reference to the start of the input string.

In figures 3.4.3 and 3.4.4 the remainder of the program execution is illustrated. Here the same call setup is done for arguments and stack pointer management. `RDI` and `RSI` are used as the source and destination index registers, passing pointers to the two argument arrays. At this point, we can be certain that vulnerability persists in the lifted code. `strcpy` does

```
00100674:
  store i64 7016996765293437281, i64* @RAX
  br label %0010067e
0010067e:
  store i64 7016996765293437281, i64* @RDX
  br label %00100688
00100688:
  %.53 = getelementptr i8*, i8** @RBP, i64 0, i64 -64
  %.54 = load i64, i64* @RAX
  %.55 = bitcast i8* %.53 to i64*
  store i64 %.54, i64* %.55
  br label %0010068c
0010068c:
  %.57 = getelementptr i8*, i8** @RBP, i64 0, i64 -56
  %.58 = load i64, i64* @RDX
  %.59 = bitcast i8* %.57 to i64*
  store i64 %.58, i64* %.59
  br label %00100690
00100690:
  %.61 = getelementptr i8*, i8** @RBP, i64 0, i64 -48
  %.62 = load i64, i64* @RAX
  %.63 = bitcast i8* %.61 to i64*
  store i64 %.62, i64* %.63
  br label %00100694
00100694:
  %.65 = getelementptr i8*, i8** @RBP, i64 0, i64 -40
  %.66 = load i64, i64* @RDX
  %.67 = bitcast i8* %.65 to i64*
  store i64 %.66, i64* %.67
  br label %00100698
00100698:
  %.69 = getelementptr i8*, i8** @RBP, i64 0, i64 -32
  %.70 = load i64, i64* @RAX
  %.71 = bitcast i8* %.69 to i64*
  store i64 %.70, i64* %.71
  br label %0010069c
0010069c:
  %.73 = getelementptr i8*, i8** @RBP, i64 0, i64 -24
  %.74 = load i64, i64* @RDX
  %.75 = bitcast i8* %.73 to i64*
  store i64 %.74, i64* %.75
  br label %001006a0
001006a0:
  %.77 = getelementptr i8*, i8** @RBP, i64 0, i64 -16
  %.78 = bitcast i8* %.77 to i16*
  store i16 24929, i16* %.78
  br label %001006a6
001006a6:
  %.80 = getelementptr i8*, i8** @RBP, i64 0, i64 -64
  %.81 = load i8, i8* %.80
  %.82 = bitcast i64* @RAX to i8*
  store i8 %.81, i8* %.82
  br label %001006aa
001006aa:
  %.84 = ptrtoint i64* @RAX to i64
  store i64 %.84, i64* @RDI
  br label %001006ad
001006ad:
  %.86 = getelementptr i8*, i8** @RSP, i64 0, i64 -8
  store i8* %.86, i8** @RSP
  %.88 = getelementptr i8*, i8** @RSP, i64 0, i64 0
  %.89 = bitcast i8* %.88 to i64*
  store i64 1050290, i64* %.89
  call void @foo()
  br label %001006b2
001006b2:
  store i64 0, i64* @RAX
  br label %001006b7
```

Figure 3.4.2: Example of a lifted LLVM IR in Ghidra-to-LLVM

not account for bounds checking, so the larger array will overflow the smaller and spill onto the stack. A malicious actor could then inject instructions and execute arbitrary code.

Figure 3.4.4 illustrates how stack is layed out when the final call to `strcpy` is made. At this point, `RAX` is loaded with a pointer to the input string, and `RDX` is loaded with a pointer to the output string. As `strcpy` does not check bounds on strings for copying, it will overflow the buffer of char array `c` and be vulnerable to exploitation.

```
00100652:
  %.45 = getelementptr i8*, i8** @RBP,
    i64 0, i64 -24
  %.46 = load i64, i64* @RDI
  %.47 = bitcast i8* %.45 to i64*
  store i64 %.46, i64* %.47
  br label %00100656
00100656:
  %.49 = getelementptr i8*, i8** @RBP,
    i64 0, i64 -24
  %.50 = load i8, i8* %.49
  %.51 = bitcast i64* @RDX to i8*
  store i8 %.50, i8* %.51
  br label %0010065a
0010065a:
  %.53 = getelementptr i8*, i8** @RBP,
    i64 0, i64 -5
  %.54 = load i8, i8* %.53
  %.55 = bitcast i64* @RAX to i8*
  store i8 %.54, i8* %.55
  br label %0010065e
0010065e:
  %.57 = load i64, i64* @RDX
  store i64 %.57, i64* @RSI
  br label %00100661
00100661:
  %.59 = load i64, i64* @RAX
  store i64 %.59, i64* @RDI
  br label %00100664
00100664:
  %.61 = getelementptr i8*, i8** @RSP,
    i64 0, i64 -8
  store i8* %.61, i8** @RSP
  %.63 = getelementptr i8*, i8** @RSP,
    i64 0, i64 0
  %.64 = bitcast i8* %.63 to i64*
  store i64 1050217, i64* %.64
  call void @strcpy()
  br label %00100669
```

Figure 3.4.3: Calling of `strcpy` from `foo`

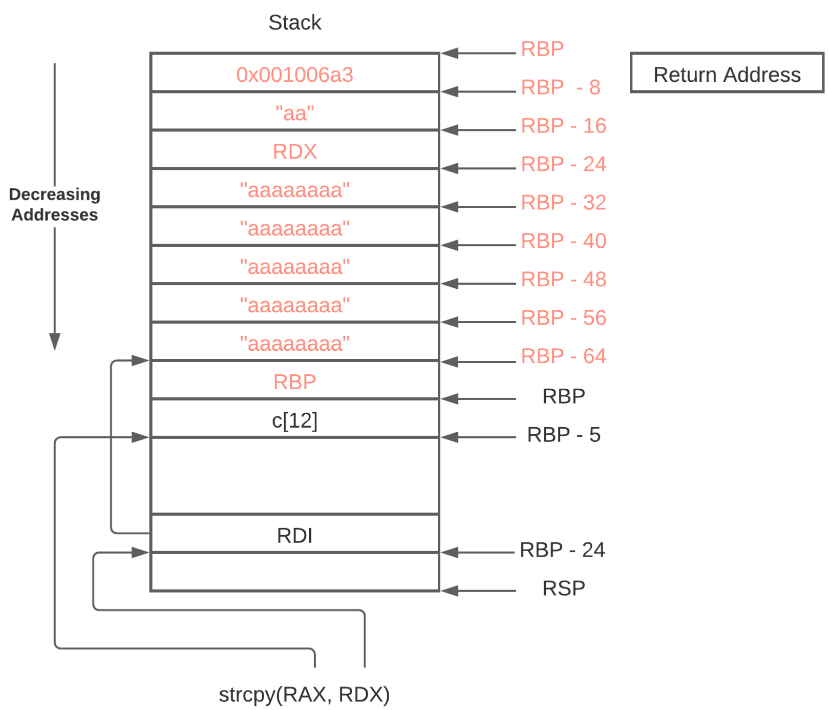

Figure 3.4.4: Calling `strcpy`

Chapter 4

Ghidrall

4.1 Overview

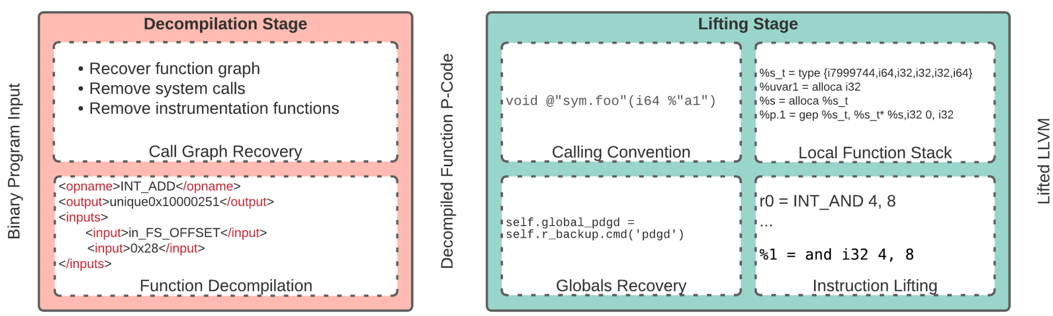

Figure 4.1.1: Overview for Ghidrall.

The main difference between Ghidrall and Ghidra-to-LLVM is that Ghidrall adds a decompilation stage instead of a disassembly stage. Figure 4.1.1 shows the overview of Ghidrall with snippets of a motivating example. The *decompilation* stage takes binary program inputs and extracts decompilation-related data structures. The *lifting* stage takes these structures and constructs the final lifted LLVM. In this chapter the decompilation and lifting stages are presented in detail, as well as an example of a buffer overflow vulnerability being preserved after lifting.

The *decompilation* stage comes in two major sub-stages: call graph recovery and function decompilation. The former selects which functions to decompile and lift. In a regular binary there are dozens of functions, however only some of these are actually used in reverse engineering. This is done creating a call graph and walking it from the entry function till the entire closed graph is explored. These functions are then decompiled using a modified version of the Ghidra decompiler using the command-line reverse engineering platform rizin. All the relevant decompilation data structures are stored in intermediate XML files.

The *lifting* stage takes in these decompilation data structures and performs a series of sub-stages to produce the lifted LLVM. Calling convention recovery is performed to validate function arguments and eliminate ambiguity between different stages of Ghidra. Local function stack recovery is performed to fix errors in the Ghidra decompiler regarding arrays and complex data structures in source code. Global recovery is performed separately as Ghidra decompiles programs function-by-function and a global set of memory is not well defined. Finally, instruction lifting is performed on each P-Code operation of each function in decompilation based on translation rules between P-Code and LLVM.

The source for Ghidrall and its test can be found at the following webpage: `https://github.com/toor-de-force/Ghidrall`.

4.2 Decompilation

Algorithm 8 shows the steps involved in the decompilation stage. Lines 3–15 illustrate the call graph recovery steps needed to determine which functions to decompile. The `DecompileFunction` procedure then takes in those functions and emits the intermediate decompiled functions for the lifter.

4.2.1 Call Graph Recovery

One challenge in automating the reverse engineering process is selecting the appropriate functions to decompile. Large programs can consist of thousands of functions, increasing complexity for both human and machine analysis.

Before decompiling individual functions in rizin, Ghidrall chooses which functions to decompile. 4.2.1 is an example output from `rizin`'s `afl` command. This lists function names and their associated addresses. From a specified entry point the complete call graph is reconstructed and then pruned to eliminate unreachable nodes. These are typically the background/system functions common across all binary programs (lines `1–6,11,14,15,19`).

Algorithm 8: Ghidrall Call Graph Recovery Algorithm

```
   Result: Decompilation Data Structures XML
1  to_visit ← [entry];
2  visited ← [ ];
3  while to_visit > 0 do
4  |  current = to_visit.pop;
5  |  foreach function reference in current do
6  |  |  if reference is instrumented then
7  |  |  |  next;
8  |  |  else if reference in system then
9  |  |  |  nextx
10 |  |  else if reference in visited then
11 |  |  |  next;
12 |  |  else
13 |  |  |  to_visit ← reference
14 |  |  end
15 |  end
16 |  visited ← current;
17 end
18 foreach visited function do
19 |  DecompileFunction;
20 end
```

Other functions, which have pre-defined implementations in Ghidrall are also not selected for decompilation. Instrumentation functions (lines `7,8,9,12,13,16,20`) are predefined as they have pre-defined behaviours and should lift the same way each time in Ghidrall. System functions (lines `8,21`) are either pre-defined or not selected for decompilation, depending on whether or not their function is necessary for analysis. All functions are paired with their addresses to allow indirect function calls to occur. The resulting list of functions is then passed to the function decompilation step.

4.2.2 Function Decompilation

The standard pseudo-C output from Ghidra strips away a lot of useful information: local variables are assumed to be separate with stack positioning removed, function declarations are sometimes inconsistent, and operations are collapsed in ways that do not necessarily

```
0x00400430    entry0
0x00400470    sym.deregister_tm_clones
0x004004a0    sym.register_tm_clones
0x004004e0    sym.__do_global_dtors_aux
0x00400510    entry.init0
0x004009c0    sym.__libc_csu_fini
0x00400520    sym.nd
0x00400420    sym.imp.time
0x00400540    sym.path_start
0x00400570    sym.path_goal
0x004009c4    sym._fini
0x004005a0    sym.path_nongoal
0x004005d0    sym.example_constrain_arg
0x00400950    sym.__libc_csu_init
0x00400460    sym._dl_relocate_static_pie
0x00400640    sym.example_counter
0x00400660    main
0x00400310    sym.foo
0x004003e0    sym._init
0x00400600    sym.example_constrain_ret
0x00400410    sym.imp.printf
```

Figure 4.2.1: Sample output of `rizin afl`

make sense. The decompiler works by taking in the P-Code output from the earlier stages and performs a series of transformation and analysis passes to decompile the program.

This annotated P-Code is then passed to a code generator to produce Ghidra's pseudo-C. This internal state is not immediately accessible and required adding algorithm 9 to extract decompilation information from the internal decompilation data structures, with some re-engineering required to maintain references to information normally lost in the decompilation passes.

4.3 Lifting Stage

Algorithm 10 illustrates the procedures used in the lifting process. First, globals are recovered from the intermediate files. Then each function has its calling convention recovered,

Algorithm 9: Ghidrall Decompilation Algorithm

```
Result: Internal Decompilation Data Structure XML
Emit basic blocks in flat structure;
Emit return type reference;
Emit function name and address;
Emit parameter stack range;
foreach parameter do
    Emit name, typeref, space and offset;
end
Emit local variable stack range;
foreach local variable do
    Emit name, typeref, space and offset;
end
foreach basic block do
    Emit id and address;
    foreach operation do
        Emit operation name;
        Emit output varnode;
        foreach input do
            Emit input varnode;
        end
    end
    Emit in and out branches;
end
```

Algorithm 10: Ghidrall Lifting Algorithm

```
Result: Valid Lifted LLVM
GlobalRecovery;
foreach function do
    CallingConventionRecovery;
    LocalFunctionStack;
    InstructionLifting;
end
```

has its local function stack built and has each of its instructions lifted.

4.3.1 Global Recovery

Global variables are not accurately maintained across functions as each function is independently decompiled. As such a list of global variables is constructed using their decompiled name as well as their address. This list is constructed by doing xpath queries across all of the intermediate files. All the references are then connected to maintain a single global variable reference list.

4.3.2 Calling Convention Recovery

Calling convention for a common function in a single program is not necessarily consistent in Ghidra. This is because each function is analyzed independently and information is not propagated across each state. For example, a common error is variability in the number of arguments in declaration and usage of the same function (`void A(param1, param2) vs void A(param1)`). Ghidrall corrects this by propagating the calling convention recovered in the function declaration across all references to the function, with some added repair steps if there is a mismatch in the number of arguments. We trust the function declaration over references as the declaration is typically closer to being accurate.

4.3.3 Local Function Stack Recovery

Ghidra's representation of local variables in functions is broken. In figure 4.4.1 the character array `bad` is broken apart into a series of 4-byte values which assume values will be adjacent to each other on compilation; C standard makes no such guarantees. Additionally, the values are annotated with what appear to be stack positions. The Ghidra decompiler maintains the concept of a stack until the pseudo-C tokens are emitted. Since Ghidra's internal data structures are consumed by Ghidrall's *lifter* and LLVM data structures support memory adjacency it is possible to perform decompilation better. The performance of the three approaches for local variable recovery are contrasted in the evaluation section.

(a) Simplistic lifting strategy

Each local variable's internal decompilation data is used to allocate the necessary amount of memory as dictated by the variable P-Code type and size using LLVM's `alloca` in sequence. For example, the value `u1` is of P-Code type `undefined4` with size 4. It becomes `%u1 = alloca i32`. This maintains the same issue in Ghidra, as each value is independent of other values and complex data structures will not be lifted correctly.

```
%u1 = alloca i32
%v18 = alloca i64
%v10 = alloca i32
%v8 = alloca i32
%v4 = alloca i64
%r0 = alloca i32
%r8 = alloca i32
%r206 = alloca i8
%r10 = alloca i32
```

(a) Simplistic lifting strategy

```
%s_t = type {i7999744,i64,i32,i32,i32,i64}

%uvar1 = alloca i32
%s = alloca %s_t
%p.1 = getelementptr %s_t , %s_t* %s,i32 0, i32 0
%v18 = getelementptr %s_t,%s_t* %s,i32 0, i32 1
%v10 = getelementptr %s_t,%s_t* %s,i32 0,i32 2
%p.2 = getelementptr %s_t,%s_t* %s,i32 0, i32 3
%v8 = getelementptr %s_t,%s_t* %s,i32 0, i32 4
%v4 = getelementptr %s_t,%s_t* %s,i32 0, i32 5
%r0 = alloca i32
%r8 = alloca i32
%r206 = alloca i8
%r10 = alloca i32
```

(b) Single struct strategy

```
%s = alloca [999999 x i8]
%u1 = alloca i32
%1 = getelementptr [999999 x i8],[999999 x i8]* %s, i32 0,i32 999968
%v18 = bitcast i8* %1 to i64*
%2 = getelementptr [999999 x i8],[999999 x i8]* %s,i32 0,i32 999976
%v10 = bitcast i8* %2 to i32*
%3 = getelementptr [999999 x i8],[999999 x i8]* %s,i32 0, i32 999984
%v8 = bitcast i8* %3 to i32*
%4 = getelementptr [999999 x i8],[999999 x i8]* %s,i32 0,i32 999988
%v4 = bitcast i8* %4 to i64*
%r0 = alloca i32
%r8 = alloca i32
%r206 = alloca i8
%r10 = alloca i32
```

(c) Byte addressable stack strategy

Figure 4.3.1: Different Formats of Local Function Stack Recovery

(b) Single struct strategy

Using a single LLVM struct to represent the local variable stack requires mapping the type of each variable in the stack as well as its address to values in a struct, while inserting padding between gaps to maintain the correctness of relative indexing. For example, in 4.3.1 `(b)`, `%s_t` is declared and then `alloca`'d to `%s`. Values `%p.1` and `%p.2` are padding values inserted to maintain positioning of variables `%v18`, `%v10`, `%v8`, `%v4`. The values are accessible by creating a pointer to the position using the `getelementptr` function in LLVM and indexing into the correct position in `%s`. The remaining variables `%uvar1`, `%r0`, `%r8`, `%r206`, and `%r10` are all recovered as independant variables as analysis passes did not find any relative indexing.

(c) Byte addressable stack strategy

For the byte addressable strategy in 4.3.1 `(c)`, the struct is replaced with a single arbitrarily large array of bytes, represented in LLVM as `%s = alloca [999999 x i8]`. Pointers to the correct index for each variable are created using the `getelementptr` instruction and then `bitcast` to the correct size and stored into the named variable.

4.3.4 Instruction Lifting

No.	P-Code	LLVM IR via llvmlite
1	B = COPY A	%temp = load i8, i8* %A store i8 %temp, i8* %B
2	val:1 = LOAD ram(addr)	%val = load i8, i8* %addr
3	STORE ram(addr), val:4	store i32 %val, i32* %addr
4	BRANCH *[ram]addr	br label %"3"
5	CBRANCH *[ram]addr, val	br i1 %val, label %1, label %"2"
6	CALL *[ram]0x8048728b	call void @sym.path_start()
7	RETURN	ret void
8	out = INT_EQUAL A, 0:4	%out = icmp eq i32 %A, 0
9	out = INT_ADD A, 3:4	%out = add i32 %A, 3
10	out:1 = BOOL_XOR A, 1:1	%out = xor i1 %A, 1
11	out = PTRSUB A,3:4	%out = gep %A, %A*, i32 0, i32 3

Table 4.3.1: Mappings of High P-Code to LLVM IR

Once the previous stages are complete, instruction lifting proceeds similarly to Ghidra-to-LLVM. P-Code operations are mapped to one or more LLVM operations with sanitization performed on the operands to bridge the rules between the two languages. Table 4.3.1

illustrates examples of mapping P-Code instructions to LLVM IR. Special instructions like `PTRSUB`, `PTRADD`, `PIECE` and `SUBPIECE` make usage of the special stack recovery as each of these instructions require relative indexing to access fields (the former two) or concatenation and selecting specific bits (the latter two).

4.4 Example

Figure 4.4.1 is a comparison of source code for a motivating buffer overflow example and the output from the standard Ghidra decompiler in pseudo-C. The vulnerability in this example comes from line `21` in `foo` of `(a)`, where a string copy is performed without bounds-checking on arrays of mismatched size. An attacker could exploit the resulting buffer overflow to perform arbitrary code execution. Sub-figure `(b)` shows the standard decompiler output from Ghidra. Figure 4.4.2 is the LLVM output from Ghidrall for the `main` function.

Each of the steps in Ghidrall addresses an issue in the standard Ghidra decompilation. Call graph recovery prunes the number of functions to lift. Function decompilation recovers more information from the decompilation than is present in the pseudo-C. Calling convention recovery propagates the function signature found in the function declaration to all references. Local function stack recovery eliminates the bizarre variable representation found 4.4.1 `(b)`. Globals recovery merges the local function scope with the global scope to maintain consistency in variable types and names. And lastly the instruction lifting maps High P-Code operations to LLVM IR for re-compilation. The following section validates the output of Ghidrall.

```c
#include <string.h>
#include <stdio.h>

char *my_strcpy(char *destination,
const char *source) {
  if (destination == NULL)
    return NULL;
  char *ptr = destination;
  while (*source != '\0') {
    *destination = *source;
    destination++;
    source++;
  }
  *destination = '\0';
  return ptr;
}


void foo(char *bar) {
  volatile char c[5];
  my_strcpy((char*)c, bar);
}

int main() {
  volatile char bad[50] =
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa";
  foo((char *)&bad);
  return 0;
}
```

(a) Source code

```c
undefined8 main(void)
{
    int64_t var_40h;
    undefined4 uStack64;
    undefined4 uStack60;
    undefined4 uStack56;
    undefined4 uStack52;
    undefined4 uStack48;
    undefined4 uStack44;
    undefined4 uStack40;
    undefined4 uStack36;
    undefined4 uStack32;
    undefined4 uStack28;
    undefined2 uStack24;
    int64_t var_4h;

    var_4h._0_4_ = 0;
    uStack40 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._32_4_;
    uStack36 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._36_4_;
    uStack32 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._40_4_;
    uStack28 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._44_4_;
    uStack56 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._16_4_;
    uStack52 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._20_4_;
    uStack48 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._24_4_;
    uStack44 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._28_4_;
    var_40h._0_4_ = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._0_4_;
    var_40h._4_4_ = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._4_4_;
    uStack64 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._8_4_;
    uStack60 = str.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa._12_4_;
    uStack24 = 0x6161;
    sym.foo((int64_t)&var_40h);
    return 0;
}
```

(b) Standard Ghidra Decompiler Output for `main`

Figure 4.4.1: Comparison of source and Ghidra decompiled C

```
define i64 @"main"()
{
"0":
  %".2" = alloca %"local_struct.main"
  %"padding" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 0
  %"var_40h" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 1
  %"uStack64" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 2
  %"uStack60" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 3
  %"uStack56" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 4
  %"uStack52" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 5
  %"uStack48" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 6
  %"uStack44" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 7
  %"uStack40" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 8
  %"uStack36" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 9
  %"uStack32" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 10
  %"uStack28" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 11
  %"uStack24" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 12
  %"padding.1" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 13
  %"var_4h" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 14
  %"register0x0" = alloca i64
  %".3" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 0
  %".4" = bitcast i8* %".3" to i32*
  %".5" = load i32, i32* %".4"
  store i32 %".5", i32* %"uStack40"
  %".7" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 4
  %".8" = bitcast i8* %".7" to i32*
  %".9" = load i32, i32* %".8"
  store i32 %".9", i32* %"uStack36"
  %".11" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 8
  %".12" = bitcast i8* %".11" to i32*
  %".13" = load i32, i32* %".12"
  store i32 %".13", i32* %"uStack32"
  %".15" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 12
  %".16" = bitcast i8* %".15" to i32*
  %".17" = load i32, i32* %".16"
  store i32 %".17", i32* %"uStack28"
  %".19" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 0
  %".20" = bitcast i8* %".19" to i32*
  %".21" = load i32, i32* %".20"
  store i32 %".21", i32* %"uStack56"
  %".23" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 4
  %".24" = bitcast i8* %".23" to i32*
  %".25" = load i32, i32* %".24"
  store i32 %".25", i32* %"uStack52"
  %".27" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 8
  %".28" = bitcast i8* %".27" to i32*
  %".29" = load i32, i32* %".28"
  store i32 %".29", i32* %"uStack48"
  %".31" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 12
  %".32" = bitcast i8* %".31" to i32*
  %".33" = load i32, i32* %".32"
  store i32 %".33", i32* %"uStack44"
  %".35" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 0
  %".36" = bitcast i8* %".35" to i32*
  %".37" = load i32, i32* %".36"
  %".38" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 4
  %".39" = bitcast i8* %".38" to i32*
  %".40" = load i32, i32* %".39"
  %".41" = zext i32 %".40" to i64
  store i64 %".41", i64* %"var_40h"
  %".43" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 8
  %".44" = bitcast i8* %".43" to i32*
  %".45" = load i32, i32* %".44"
  store i32 %".45", i32* %"uStack64"
  %".47" = getelementptr [17 x i8], [17 x i8]* @"0x00400620", i32 0, i32 12
  %".48" = bitcast i8* %".47" to i32*
  %".49" = load i32, i32* %".48"
  store i32 %".49", i32* %"uStack60"
  store i16 24929, i16* %"uStack24"
  %".52" = ptrtoint i64* %"var_40h" to i64
  call void @"sym.foo"(i64 %".52")
  store i64 0, i64* %"register0x0"
  %".55" = load i64, i64* %"register0x0"
  ret i64 %".55"
}
```

Figure 4.4.2: Ghidrall LLVM output for `main`

Chapter 5

Evaluation

5.1 Simple Password Challenge

```c
int main() {
    int a = INT_RAND;
    int b = INT_RAND;
    int c = INT_RAND;
    int d = INT_RAND;
    int e = INT_RAND;
    int f = INT_RAND;
    int g = INT_RAND;
    if (a<97||a>122) return 1;
    if (b<97||b>122) return 1;
    if (c<97||c>122) return 1;
    if (d<97||d>122) return 1;
    if (e<97||e>122) return 1;
    if (f<97||f>122) return 1;
    if (g<97||g>122) return 1;
    if ((char)(((((a*32)>>2)%26)+65) != 'C')) return 1;
    if ((char)(((((b*23)>>2)%26)+65) != 'I')) return 1;
    if ((char)(((((c*22)>>2)%26)+65) != 'Z')) return 1;
    if ((char)(((((d*42)>>2)%26)+65) != 'U')) return 1;
    if ((char)(((((e*15)>>2)%26)+65) != 'L')) return 1;
    if ((char)(((((f*25)>>2)%26)+65) != 'Q')) return 1;
    if ((char)(((((g*29)>>2)%26)+65) != 'E')) return 1;
    path_goal();
    return 0;
}
```

Figure 5.1.1: A Simple Password Challenge

Figure 5.1.1 is a challenge that was created for undergraduate students. These types of challenges are typically designed by security practitioners and based on real-world encounters to test and engage other engineers. When compiled, this binary takes a single command line argument as a password and compares it against a series of checks to verify if the password is correct. In this example, the intended password is "`reverse`".

Ghidrall	SeaHorn	Description
INT_RAND	nd()	Non-deterministic input
path_start()	-	Program start
path_goal()	verifier.error()	Program objective
path_non_goal()	verifier.error()	Program failure state

Figure 5.1.2: Ghidrall and SeaHorn Instrumentation Functions

Table 5.1.2 lists the functions that are used by Ghidrall and SeaHorn to solve problems. In this example, the source code is already instrumented with Ghidrall and SeaHorn functions to automate lifting and solving. Program inputs are replaced character-by-character with the output of `INT_RAND`, which maps to SeaHorn's `nd` function. This treats the output as non-deterministic so SeaHorn knows to perform its solving based on these variables. Ghidrall maps the `path_goal` function to `verifier.error` in LLVM IR. SeaHorn then attempts to solve the problem by either proving the goal is unreachable or by providing a counterexample. The lifted LLVM IR can be found in Appendix C.

Running this through SeaHorn, we find that another password is possible. SeaHorn's provided counter example "`@0 = private constant [7 x i32] [i32 114, i32 119, i32 118, i32 101, i32 121, i32 115, i32 101]`" looks like an array of ASCII values. This translates to the string "`enveysw`". Plugging this back into the original binary, we find that this password also works. Ghidrall and Seahorn are collectively able to solve this password problem automatically, and provided us with an unexpected alternative solution.

5.2 Functional Verification

5.2.1 Test Generation

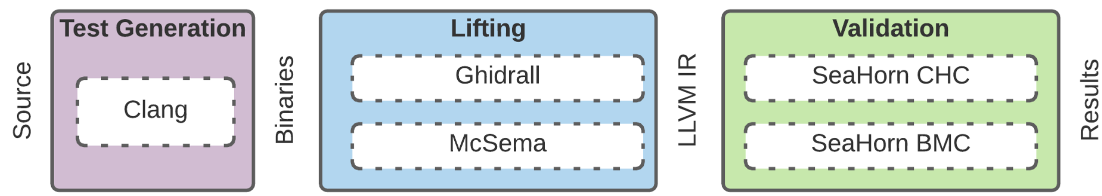

Figure 5.2.1: Test Generation

Figure 5.2.1 illustrates the evaluation procedure for this section. 97 different programs were used to verify the functional accuracy of the lifters under test. Two major set of results were extracted — the performance of each of Ghidrall's function stacks against one another, and the performance of the best Ghidrall mode against another lifter, McSema. All lifted results were then passed through SeaHorn with the expected results to validate their functional accuracy in reaching goals and nongoals.

Figure 5.2.2 is an example of a verification problem that was used to validate Ghidrall's lifting with SeaHorn. Each of these problems include a non-deterministic input (`INT_RAND`) as well as a goal (`path_goal()` and a non-goal (`path_nongoal()`). Two versions are then compiled for each program for each of the goal states.

5.2.2 Comparing Stack Structures

Stack Format	Passes	Fails	Lifting Fails	Timeouts	Success Rate (%)
no_option	484	64	30	4	83.16
byte_addressable	483	87	8	4	82.99
single_struct	501	67	8	6	86.08

Table 5.2.1: Success Rate Overall for Structures

Table 5.2.1 shows the success rate for each of the local function stack options. There are two main interesting findings. Firstly, any test that involves data structures fails in the lifting

```

#include "test.hpp"


int main() {
  path_start();
  int n = INT_RAND;
  volatile int x = n;
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      path_goal();
      if (i > x) {
        path_nongoal();
      }
    }
  }
}
```

Figure 5.2.2: Example Functional Verification Problem

stage with the `no_option` stack structure. 22 tests failed there, with the primary reason being that later P-Code and LLVM IR instructions attempt to index or access those data structures in ways that do not make sense. For instance, the data may be accessed under the assumption that sequentially defined data is arranged in the same order in memory. Those types of accesses are only valid in both the `byte_addressable` and `single_struct` stack structures. The second interesting finding is that the `single_struct` stack structure has the best success rate, while `byte_addressable` is actually the worst. The improvement in lifting failures for the former translates to an improved overall success rate making it the best option for future decompiling lifter designs.

Table 5.2.2 breaks down the results by test type. For each of the 97 test programs tests are generated at three different compilation optimization modes and with goal and non-goal settings as SeaHorn tests these separately.

Stack Format	Optimization	Goal	Pass	Fail	Lifting Fail	Timeout
no_option	0	goal	80	10	7	0
		nongoal	74	16	7	0
	1	goal	79	12	5	1
		nongoal	85	6	5	1
	2	goal	81	12	3	1
		nongoal	85	8	3	1
byte_addressable	0	goal	84	12	1	0
		nongoal	84	12	1	0
	1	goal	73	21	2	1
		nongoal	84	10	2	1
	2	goal	74	21	1	1
		nongoal	84	11	1	1
single_struct	0	goal	84	11	1	1
		nongoal	78	17	1	1
	1	goal	82	12	2	1
		nongoal	87	7	2	1
	2	goal	83	12	1	1
		nongoal	87	8	1	1

Table 5.2.2: Comparison Local Function Stacks

5.2.3 Comparing Lifters

Lifter	Passes	Fails	Lifting Fails	Timeouts	Success (%)	Avg. LOC
Ghidrall	501	67	8	6	86.08	61.1
McSema	414	168	0	0	71.13	3417.6

Table 5.2.3: Overall Success Rate of Lifters

Table 5.2.3 compares the best performing stack structure of Ghidrall (`single_struct`) and McSema with DysInst as its CFG generator. DysInst was used instead of McSema's main option of IdaPro due to license costs. These results show Ghidrall performing better than McSema by a large margin. The reduction in file length is also significant as it illustrates the difference between the motivations of the two programs. McSema is a disassembling lifter that prioritizing re-compilation, where Ghidrall is a decompiling lifter that prioritizes readability and usefulness from a higher-level in the reverse engineering process.

Chapter 6

Related Work

6.1 Lifters

McSema[13] is an executable LLVM lifter produced by Trail of Bits. It preserves programs such that they may be re-compiled, so the end result is LLVM IR that is closer to machine instructions than higher level decompilation[18]. McSema has a two step process to produce LLVM IR. The first step is control flow recovery, which requires disassemblers like Ida Pro. The second step is instruction translating, which maps machine instructions to LLVM IR through the Remill library[19]. The version of McSema that Ghidrall is tested against uses DynInst for the first stage[20]. McSema has a few interesting features that Ghidrall does not. For instance, it handles C++ exceptions[21]. This can be a difficult challenge due to features like runtime errors. It does this by emulating how exceptions are handled in Linux systems. McSema associates exception handlers and cleanup methods with blocks that raise exceptions and uses the LLVM `invoke` instruction to call them. Ghidrall is currently unable to replicate this feature.

McToll[14][22] is a lifter released by Microsoft. It shares some features that Ghidrall does, such as function prototype discovery and stack frame recovery. Like McSema, it too includes features for C++ like vtables, name mangling and exception handling. McToll also is structured to be re-compilable like McSema. One main limitation it has is a requirement to annotate each lifted program with a list of functions to include/exclude as well as pointing to library functions.

RetDec[15][23] is a complete decompiler and lifter from Avast. RetDec has similar features as McSema and McToll in that it can manage C++ features and is also designed to be re-targetable. It is also capable of recovering debugging information in binaries. RetDec can also emit human-readable code in either a C-like or Python-like pseudocode.

6.2 Pharos

The **Pharos Static Binary Analysis Framework**[24][25], produced by Carnegie Mellon's Software Engineering Institute, is a series of reverse engineering analysis tools built using the ROSE compiler infrastructure[26]. It performs static analysis, control flow analysis and dataflow analysis. The functionality test-set used by Ghidrall was produced by the Pharos team. Pharos consists of the following tools:

- APIAnalyzer: A tool for finding API Calls within a binary like common operating system calls.
- OOAnalyzer: A tool for recovering object-oriented code.
- CallAnalyzer: A tool for recovering static parameters of function calls.
- FN2Yara: A tool to generate YARA signatures from functions.
- FN2Hash: A tool for generating useful hashes from binaries.
- DumpMASM: A tool for dumping assembly from a binary.

Pharos can be used to find paths to interesting execution states for malicious binary reverse engineering[27]. They use constraint-based analysis with the Z3-theorem prover to generate constraints and to find paths of interest. More recent work has involved using Satisfiability Modulo Theorem (SMT) to create a new tool called **GhiHorn**[28]. This strategy is similar to how Ghidrall uses SeaHorn, in that it is able to determine whether or not a path is reachable and if not, prove why it is unreachable. GhiHorn translates P-Code directly to SMT-Lib format for horn clauses.

Chapter 7

Conclusion

This thesis presents two reverse engineering tools to enable automated reverse engineering and vulnerability discovery: the disassembling lifter Ghidra-to-LLVM and the decompiling lifter Ghidrall. Both types of lifters have their benefits, and illustrate that it is possible to lift binaries to LLVM IR with further decompilation and preserve program functionality. Additionally, features of decompiling lifters like function stack recovery are shown to be critical to preserving behaviour.

In the future further work is necessary to discover better decompilation strategies and how they can enable vulnerability researchers to discover vulnerabilities at lower cost in time and resources. Additionally, greater support in Ghidrall is needed to make it a more mature tool for use. For example, features like heap representations and variable function arguments are currently missing.

References

[1] T. Cipresso and M. Stamp, *Software Reverse Engineering*, pp. 659–696. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[2] D. Kushner, "The real story of stuxnet," *IEEE Spectrum*, vol. 50, no. 3, pp. 48–53, 2013.

[3] C. Eagle, *The IDA pro book*. no starch press, 2011.

[4] D. Brumley, "The cyber grand challenge and the future of cyber-autonomy," *USENIX Login*, vol. 43, no. 2, pp. 6–9, 2018.

[5] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, "The Mayhem Cyber Reasoning System," *IEEE Security Privacy*, vol. 16, no. 2, pp. 52–60, 2018.

[6] A. Nguyen-Tuong, D. Melski, J. W. Davidson, M. Co, W. Hawkins, J. D. Hiser, D. Morris, D. Nguyen, and E. Rizzi, "Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge," *IEEE Security Privacy*, vol. 16, no. 2, pp. 42–51, 2018.

[7] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, *et al.*, "Mechanical phish: Resilient autonomous hacking," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 12–22, 2018.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[9] M. Zalewski, "Google/afl: American fuzzy lop - a security-oriented fuzzer," 2020.

[10] LLVM Project, "Llvm Language Reference Manual," 2022.

[11] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (USA), p. 209–224, USENIX Association, 2008.

[12] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn Verification Framework," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), (Cham), pp. 343–361, Springer International Publishing, 2015.

[13] Trail of Bits, "lifting-bits/mcsema: Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode," 2015.

[14] Microsoft, "microsoft/llvm-mctoll: llvm-mctoll," 2019.

[15] Avast, "avast/retdec: RetDec is a retargetable machine-code decompiler based on LLVM," 2018.

[16] National Security Agency, "Ghidra," 2019.

[17] National Security Agency, "P-code Reference Manual," Sep 2017.

[18] Trail of Bits, "Heavy lifting with McSema 2.0," Jan 2018.

[19] Trail of Bits, "lifting-bits/remill: Library for lifting of x86, amd64, and aarch64 machine code to LLVM bitcode," 2015.

[20] L. KORENČIK, "Decompiling binaries into llvm ir using mcsema and dyninst [online]," master's thesis, Masaryk University, Faculty of Informatics, Brno, 2019 [cit. 2021-12-02].

[21] Trail of Bits, "How McSema Handles C Exceptions," Jan 2019.

[22] S. B. Yadavalli and A. Smith, "Raising Binaries to LLVM IR with MCTOLL (WIP Paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, (New York, NY, USA), p. 213–218, Association for Computing Machinery, 2019.

[23] J. Křoustek and P. Matula, "RetDec: An Open-Source Machine-Code Decompiler." [talk], July 2018. Presented at Pass the SALT 2018, Lille, FR.

[24] CMU-SEI, “cmu-sei/pharos: Automated static analysis tools for binary programs,” 2017.

[25] J. Gennari, “Pharos Binary Static Analysis Tools Released on GitHub,” Aug 2017.

[26] D. Quinlan and C. Liao, “The ROSE source-to-source compiler infrastructure,” in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011, p. 1, Citeseer, 2011.

[27] J. Gennari, “Path Finding in Malicious Binaries: First in a Series,” Dec 2018.

[28] J. Gennari, “Ghihorn: Path Analysis in Ghidra Using SMT Solvers,” Oct 2021.

APPENDICES

Appendix A

Ghidra-to-LLVM's bof.c

```
#include <string.h>

// Compiled with: clang -fno-stack-protector buffer_overflow.c -o
// buffer_overflow.bin

#include <stdio.h>

// Function to implement strcpy() function
char *my_strcpy(char *destination, const char *source) {
  // return if no memory is allocated to the destination
  if (destination == NULL)
    return NULL;

  // take a pointer pointing to the beginning of destination string
  char *ptr = destination;

  // copy the C-string pointed by source into the array
  // pointed by destination
  while (*source != '\0') {
    *destination = *source;
    destination++;
    source++;
  }

  // include the terminating null character
  *destination = '\0';

  // destination is returned by standard strcpy()
  return ptr;
}


void foo(char *bar) {
  volatile char c[5];
  my_strcpy((char*)c, bar); // no bounds checking
}

int main() {
  volatile char bad[50] =
  "aaaaaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaaaaa";
  foo((char *)&bad);
  return 0;
}
```

Appendix B

Ghidra-to-LLVM's vuln.ll

```
; ModuleID = lifted
target triple = x86_64-pc-linux-gnu
target datalayout = e-m:e-p:32:32-f64
    :32:64-f80:32-n8:16:32-S128

@CF = internal global i1 0
@RSP = internal global i8* null
@OF = internal global i1 0
@SF = internal global i1 0
@ZF = internal global i1 0
@PF = internal global i1 0
@RAX = internal global i64 0
@RIP = internal global i8* null
@EBP = internal global i8* null
@RBP = internal global i8* null
@R9 = internal global i64 0
@RDX = internal global i64 0
@RSI = internal global i64 0
@R8 = internal global i64 0
@RCX = internal global i64 0
@RDI = internal global i64 0
@R15 = internal global i64 0
@R14 = internal global i64 0
@R13 = internal global i64 0
@R12 = internal global i64 0
@RBX = internal global i64 0
@R13D = internal global i32 0
@EDI = internal global i32 0
@EBX = internal global i32 0
@A_00300fe8:8 = internal global i64 0
@A_00100502:8 = internal global i64 0
@A_00300fc0:8 = internal global i64 0
@A_00300fc8:8 = internal global i64 0
@A_00300fd0:8 = internal global i64 0
@A_00100510:8 = internal global i64 0
@A_00300ff8:8 = internal global i64 0
@A_00300fe0:8 = internal global i64 0
@A_0010056a:8 = internal global i64 0
@A_001005a0:8 = internal global i64 0
@A_00300fd8:8 = internal global i64 0
@A_001005f0:8 = internal global i64 0
@A_00300ff0:8 = internal global i64 0
@A_00301010:1 = internal global i8 0
@A_00100638:8 = internal global i64 0
@A_00100623:8 = internal global i64 0
@A_00301008:8 = internal global i64 0
@A_00100530:8 = internal global i64 0
@A_00100570:8 = internal global i64 0
@A_001005b0:8 = internal global i64 0
@A_00100520:8 = internal global i64 0
@A_0010064a:8 = internal global i64 0
@A_001004f0:8 = internal global i64 0
@A_00100716:8 = internal global i64 0
@A_00100700:8 = internal global i64 0
declare void @strcpy()

define void @main()
{
0010066c:
  %.20 = ptrtoint i8** @RBP to i64
  %.21 = getelementptr i8*, i8** @RSP,
    i64 0, i64 -8
  store i8* %.21, i8** @RSP
  %.23 = getelementptr i8*, i8** @RSP,
    i64 0, i64 0
  %.24 = bitcast i8* %.23 to i64*
  store i64 %.20, i64* %.24
  br label %0010066d
0010066d:
  %.26 = ptrtoint i8** @RSP to i64
```

```
  %.27 = getelementptr i8*, i8** @RBP, i64 0, i64 0
  %.28 = bitcast i8* %.27 to i64*
  store i64 %.26, i64* %.28
  br label %00100670
00100670:
  %.30 = ptrtoint i8** @RSP to i64
  %.31 = icmp ult i64 %.30, 64
  store i1 %.31, i1* @CF
  %.33 = ptrtoint i8** @RSP to i64
  %.34 = call {i64, i1} @llvm.sadd.with.overflow.i64(i64 %.33, i64 64)
  %.35 = extractvalue {i64, i1} %.34, 1
  store i1 %.35, i1* @OF
  %.37 = getelementptr i8*, i8** @RSP, i64 0, i64 -64
  store i8* %.37, i8** @RSP
  %.39 = ptrtoint i8** @RSP to i64
  %.40 = icmp slt i64 %.39, 0
  store i1 %.40, i1* @SF
  %.42 = ptrtoint i8** @RSP to i64
  %.43 = icmp eq i64 %.42, 0
  store i1 %.43, i1* @ZF
  %.45 = ptrtoint i8** @RSP to i64
  %.46 = call i64 @llvm.ctpop.i64(i64 %.45)
  %.47 = zext i8 1 to i64
  %.48 = and i64 %.46, %.47
  %.49 = trunc i64 %.48 to i1
  store i1 %.49, i1* @PF
  br label %00100674
00100674:
  store i64 7016996765293437281, i64* @RAX
  br label %0010067e
0010067e:
  store i64 7016996765293437281, i64* @RDX
  br label %00100688
00100688:
  %.53 = getelementptr i8*, i8** @RBP, i64 0, i64 -64
  %.54 = load i64, i64* @RAX
  %.55 = bitcast i8* %.53 to i64*
  store i64 %.54, i64* %.55
  br label %0010068c
0010068c:
  %.57 = getelementptr i8*, i8** @RBP, i64 0, i64 -56
  %.58 = load i64, i64* @RDX
  %.59 = bitcast i8* %.57 to i64*
  store i64 %.58, i64* %.59
  br label %00100690
00100690:
  %.61 = getelementptr i8*, i8** @RBP, i64 0, i64 -48
  %.62 = load i64, i64* @RAX
  %.63 = bitcast i8* %.61 to i64*
  store i64 %.62, i64* %.63
  br label %00100694
00100694:
  %.65 = getelementptr i8*, i8** @RBP, i64 0, i64 -40
  %.66 = load i64, i64* @RDX
  %.67 = bitcast i8* %.65 to i64*
  store i64 %.66, i64* %.67
  br label %00100698
00100698:
  %.69 = getelementptr i8*, i8** @RBP, i64 0, i64 -32
  %.70 = load i64, i64* @RAX
  %.71 = bitcast i8* %.69 to i64*
  store i64 %.70, i64* %.71
  br label %0010069c
0010069c:
  %.73 = getelementptr i8*, i8** @RBP, i64 0, i64 -24
  %.74 = load i64, i64* @RDX
  %.75 = bitcast i8* %.73 to i64*
  store i64 %.74, i64* %.75
  br label %001006a0
001006a0:
  %.77 = getelementptr i8*, i8** @RBP, i64 0, i64 -16
  %.78 = bitcast i8* %.77 to i16*
  store i16 24929, i16* %.78
  br label %001006a6
001006a6:
  %.80 = getelementptr i8*, i8** @RBP, i64 0, i64 -64
  %.81 = load i8, i8* %.80
  %.82 = bitcast i64* @RAX to i8*
  store i8 %.81, i8* %.82
  br label %001006aa
001006aa:
  %.84 = ptrtoint i64* @RAX to i64
  store i64 %.84, i64* @RDI
  br label %001006ad
001006ad:
  %.86 = getelementptr i8*, i8** @RSP, i64 0, i64 -8
  store i8* %.86, i8** @RSP
  %.88 = getelementptr i8*, i8** @RSP, i64 0, i64 0
  %.89 = bitcast i8* %.88 to i64*
  store i64 1050290, i64* %.89
  call void @foo()
  br label %001006b2
001006b2:
  store i64 0, i64* @RAX
  br label %001006b7
001006b7:
  %.93 = ptrtoint i8** @RBP to i64
```

```
  %.94 = getelementptr i8*, i8** @RSP, i64 0, i64 0
  %.95 = bitcast i8* %.94 to i64*
  store i64 %.93, i64* %.95
  %.97 = load i8*, i8** @RSP
  store i8* %.97, i8** @RBP
  %.99 = getelementptr i8*, i8** @RSP, i64 0, i64 8
  store i8* %.99, i8** @RSP
  br label %001006b8
001006b8:
  %.101 = load i8*, i8** @RSP
  store i8* %.101, i8** @RIP
  %.103 = getelementptr i8*, i8** @RSP, i64 0, i64 8
  store i8* %.103, i8** @RSP
  ret void
}

define void @foo()
{
0010064a:
  %.14 = ptrtoint i8** @RBP to i64
  %.15 = getelementptr i8*, i8** @RSP, i64 0, i64 -8
  store i8* %.15, i8** @RSP
  %.17 = getelementptr i8*, i8** @RSP, i64 0, i64 0
  %.18 = bitcast i8* %.17 to i64*
  store i64 %.14, i64* %.18
  br label %0010064b
0010064b:
  %.20 = ptrtoint i8** @RSP to i64
  %.21 = getelementptr i8*, i8** @RBP, i64 0, i64 0
  %.22 = bitcast i8* %.21 to i64*
  store i64 %.20, i64* %.22
  br label %0010064e
0010064e:
  %.24 = ptrtoint i8** @RSP to i64
  %.25 = icmp ult i64 %.24, 32
  store i1 %.25, i1* @CF
  %.27 = ptrtoint i8** @RSP to i64
  %.28 = call {i64, i1} @llvm.sadd.with.overflow.i64(i64 %.27, i64 32)
  %.29 = extractvalue {i64, i1} %.28, 1
  store i1 %.29, i1* @OF
  %.31 = getelementptr i8*, i8** @RSP, i64 0, i64 -32
  store i8* %.31, i8** @RSP
  %.33 = ptrtoint i8** @RSP to i64
  %.34 = icmp slt i64 %.33, 0
  store i1 %.34, i1* @SF
  %.36 = ptrtoint i8** @RSP to i64
  %.37 = icmp eq i64 %.36, 0
  store i1 %.37, i1* @ZF
  %.39 = ptrtoint i8** @RSP to i64
  %.40 = call i64 @llvm.ctpop.i64(i64 %.39)
  %.41 = zext i8 1 to i64
  %.42 = and i64 %.40, %.41
  %.43 = trunc i64 %.42 to i1
  store i1 %.43, i1* @PF
  br label %00100652
00100652:
  %.45 = getelementptr i8*, i8** @RBP, i64 0, i64 -24
  %.46 = load i64, i64* @RDI
  %.47 = bitcast i8* %.45 to i64*
  store i64 %.46, i64* %.47
  br label %00100656
00100656:
  %.49 = getelementptr i8*, i8** @RBP, i64 0, i64 -24
  %.50 = load i8, i8* %.49
  %.51 = bitcast i64* @RDX to i8*
  store i8 %.50, i8* %.51
  br label %0010065a
0010065a:
  %.53 = getelementptr i8*, i8** @RBP, i64 0, i64 -5
  %.54 = load i8, i8* %.53
  %.55 = bitcast i64* @RAX to i8*
  store i8 %.54, i8* %.55
  br label %0010065e
0010065e:
  %.57 = load i64, i64* @RDX
  store i64 %.57, i64* @RSI
  br label %00100661
00100661:
  %.59 = load i64, i64* @RAX
  store i64 %.59, i64* @RDI
  br label %00100664
00100664:
  %.61 = getelementptr i8*, i8** @RSP, i64 0, i64 -8
  store i8* %.61, i8** @RSP
  %.63 = getelementptr i8*, i8** @RSP, i64 0, i64 0
  %.64 = bitcast i8* %.63 to i64*
  store i64 1050217, i64* %.64
  call void @strcpy()
  br label %00100669
00100669:
  br label %0010066a
0010066a:
  %.67 = ptrtoint i8** @RBP to i64
  %.68 = getelementptr i8*, i8** @RSP, i64 0, i64 0
  %.69 = bitcast i8* %.68 to i64*
  store i64 %.67, i64* %.69
  %.71 = load i8*, i8** @RSP
  store i8* %.71, i8** @RBP
```

```
  %.73 = getelementptr i8*, i8** @RSP, i64 0, i64 8
  store i8* %.73, i8** @RSP
  br label %0010066b
0010066b:
  %.75 = load i8*, i8** @RSP
  store i8* %.75, i8** @RIP
  %.77 = getelementptr i8*, i8** @RSP, i64 0, i64 8
  store i8* %.77, i8** @RSP
  ret void
}

declare {i64, i1} @llvm.sadd.with.overflow.i64(i64 %.1, i64 %.2)

declare i64 @llvm.ctpop.i64(i64 %.1)
```

Appendix C

Output from Password Challenge

```
; ModuleID = "/tmp/examples/password.bin"
target triple = "i386-pc-linux-gnu"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"

%"local_struct.main" = type {i7999904, i64}
declare i32 @"nd"()

declare void @"verifier.error"()

define void @"sym.path_goal"()
{
entry:
  call void @"verifier.error"()
  ret void
}

@"reloc.__libc_start_main" = global i32 0
@"segment.GNU_STACK" = global i32 0
@"sym..bss" = global i32 0
@"obj.global_time" = global i32 0
@"reloc.time" = global i32 0
@"segment.LOAD1" = global i32 0
@"obj.__ctr" = global i32 0
@"reloc.__gmon_start" = global i32 0
define i32 @"main"()
{
"0":
  %"iVar1" = alloca i32
  %"iVar2" = alloca i32
  %"iVar3" = alloca i32
  %"iVar4" = alloca i32
  %"iVar5" = alloca i32
  %"iVar6" = alloca i32
  %"iVar7" = alloca i32
  %".2" = alloca %"local_struct.main"
  %"padding" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 0
  %"var_4h" = getelementptr inbounds %"local_struct.main", %"local_struct.main"* %".2", i32 0, i32 1
  %"register0x8" = alloca i32
  %"register0x206" = alloca i8
  %"register0x0" = alloca i32
  %".3" = call i32 @"nd"()
  store i32 %".3", i32* %"iVar1"
  %".5" = call i32 @"nd"()
  store i32 %".5", i32* %"iVar2"
  %".7" = call i32 @"nd"()
  store i32 %".7", i32* %"iVar3"
  %".9" = call i32 @"nd"()
  store i32 %".9", i32* %"iVar4"
  %".11" = call i32 @"nd"()
  store i32 %".11", i32* %"iVar5"
  %".13" = call i32 @"nd"()
  store i32 %".13", i32* %"iVar6"
  %".15" = call i32 @"nd"()
  store i32 %".15", i32* %"iVar7"
  %".17" = load i32, i32* %"iVar1"
  %".18" = icmp slt i32 %".17", 97
  br i1 %".18", label %"23", label %"1"
"1":
  %".20" = load i32, i32* %"iVar1"
  %".21" = icmp slt i32 122, %".20"
```

```
  br i1 %".21", label %"23", label %"2"
"2":
  %".23" = load i32, i32* %"iVar2"
  %".24" = icmp slt i32 %".23", 97
  br i1 %".24", label %"22", label %"3"
"3":
  %".26" = load i32, i32* %"iVar2"
  %".27" = icmp slt i32 122, %".26"
  br i1 %".27", label %"22", label %"4"
"4":
  %".29" = load i32, i32* %"iVar3"
  %".30" = icmp slt i32 %".29", 97
  br i1 %".30", label %"21", label %"5"
"5":
  %".32" = load i32, i32* %"iVar3"
  %".33" = icmp slt i32 122, %".32"
  br i1 %".33", label %"21", label %"6"
"6":
  %".35" = load i32, i32* %"iVar4"
  %".36" = icmp slt i32 %".35", 97
  br i1 %".36", label %"20", label %"7"
"7":
  %".38" = load i32, i32* %"iVar4"
  %".39" = icmp slt i32 122, %".38"
  br i1 %".39", label %"20", label %"8"
"8":
  %".41" = load i32, i32* %"iVar5"
  %".42" = icmp slt i32 %".41", 97
  br i1 %".42", label %"1f", label %"9"
"9":
  %".44" = load i32, i32* %"iVar5"
  %".45" = icmp slt i32 122, %".44"
  br i1 %".45", label %"1f", label %"a"
a:
  %".47" = load i32, i32* %"iVar6"
  %".48" = icmp slt i32 %".47", 97
  br i1 %".48", label %"1e", label %"b"
b:
  %".50" = load i32, i32* %"iVar6"
  %".51" = icmp slt i32 122, %".50"
  br i1 %".51", label %"1e", label %"c"
c:
  %".53" = load i32, i32* %"iVar7"
  %".54" = icmp slt i32 %".53", 97
  br i1 %".54", label %"1d", label %"d"
d:
  %".56" = load i32, i32* %"iVar7"
  %".57" = icmp slt i32 122, %".56"
  br i1 %".57", label %"1d", label %"e"
e:
  %".59" = load i32, i32* %"iVar1"
  %".60" = shl i32 %".59", 5
  store i32 %".60", i32* %"register0x8"
  %".62" = load i32, i32* %"register0x8"
  %".63" = ashr i32 %".62", 2
  store i32 %".63", i32* %"register0x8"
  %".65" = load i32, i32* %"register0x8"
  %".66" = srem i32 %".65", 26
  %".67" = icmp eq i32 %".66", 2
  %".68" = zext i1 %".67" to i8
  store i8 %".68", i8* %"register0x206"
  %".70" = load i8, i8* %"register0x206"
  %".71" = trunc i8 %".70" to i1
  br i1 %".71", label %"f", label %"1c"
f:
  %".73" = load i32, i32* %"iVar2"
  %".74" = mul i32 %".73", 23
  store i32 %".74", i32* %"register0x8"
  %".76" = load i32, i32* %"register0x8"
  %".77" = ashr i32 %".76", 2
  store i32 %".77", i32* %"register0x8"
  %".79" = load i32, i32* %"register0x8"
  %".80" = srem i32 %".79", 26
  %".81" = icmp eq i32 %".80", 8
  %".82" = zext i1 %".81" to i8
  store i8 %".82", i8* %"register0x206"
  %".84" = load i8, i8* %"register0x206"
  %".85" = trunc i8 %".84" to i1
  br i1 %".85", label %"10", label %"1b"
"10":
  %".87" = load i32, i32* %"iVar3"
  %".88" = mul i32 %".87", 22
  store i32 %".88", i32* %"register0x8"
  %".90" = load i32, i32* %"register0x8"
  %".91" = ashr i32 %".90", 2
  store i32 %".91", i32* %"register0x8"
  %".93" = load i32, i32* %"register0x8"
  %".94" = srem i32 %".93", 26
  %".95" = icmp eq i32 %".94", 25
  %".96" = zext i1 %".95" to i8
  store i8 %".96", i8* %"register0x206"
  %".98" = load i8, i8* %"register0x206"
  %".99" = trunc i8 %".98" to i1
  br i1 %".99", label %"11", label %"1a"
"11":
  %".101" = load i32, i32* %"iVar4"
  %".102" = mul i32 %".101", 42
  store i32 %".102", i32* %"register0x8"
  %".104" = load i32, i32* %"register0x8"
  %".105" = ashr i32 %".104", 2
  store i32 %".105", i32* %"register0x8
```

```
"
  %".107" = load i32, i32* %"register0x8"
  %".108" = srem i32 %".107", 26
  %".109" = icmp eq i32 %".108", 20
  %".110" = zext i1 %".109" to i8
  store i8 %".110", i8* %"register0x206"
  %".112" = load i8, i8* %"register0x206"
  %".113" = trunc i8 %".112" to i1
  br i1 %".113", label %"12", label %"19"
"12":
  %".115" = load i32, i32* %"iVar5"
  %".116" = mul i32 %".115", 15
  store i32 %".116", i32* %"register0x8"
  %".118" = load i32, i32* %"register0x8"
  %".119" = ashr i32 %".118", 2
  store i32 %".119", i32* %"register0x8"
  %".121" = load i32, i32* %"register0x8"
  %".122" = srem i32 %".121", 26
  %".123" = icmp eq i32 %".122", 11
  %".124" = zext i1 %".123" to i8
  store i8 %".124", i8* %"register0x206"
  %".126" = load i8, i8* %"register0x206"
  %".127" = trunc i8 %".126" to i1
  br i1 %".127", label %"13", label %"18"
"13":
  %".129" = load i32, i32* %"iVar6"
  %".130" = mul i32 %".129", 25
  store i32 %".130", i32* %"register0x8"
  %".132" = load i32, i32* %"register0x8"
  %".133" = ashr i32 %".132", 2
  store i32 %".133", i32* %"register0x8"
  %".135" = load i32, i32* %"register0x8"
  %".136" = srem i32 %".135", 26
  %".137" = icmp eq i32 %".136", 16
  %".138" = zext i1 %".137" to i8
  store i8 %".138", i8* %"register0x206"
  %".140" = load i8, i8* %"register0x206"
  %".141" = trunc i8 %".140" to i1
  br i1 %".141", label %"14", label %"17"
"14":
  %".143" = load i32, i32* %"iVar7"
  %".144" = mul i32 %".143", 29
  store i32 %".144", i32* %"register0x8"
  %".146" = load i32, i32* %"register0x8"
  %".147" = ashr i32 %".146", 2
  store i32 %".147", i32* %"register0x8"
  %".149" = load i32, i32* %"register0x8"
  %".150" = srem i32 %".149", 26
  %".151" = icmp eq i32 %".150", 4
  %".152" = zext i1 %".151" to i8
  store i8 %".152", i8* %"register0x206"
  %".154" = load i8, i8* %"register0x206"
  %".155" = trunc i8 %".154" to i1
  br i1 %".155", label %"15", label %"16"
"15":
  call void @"sym.path_goal"()
  br label %"24"
"16":
  br label %"24"
"17":
  br label %"24"
"18":
  br label %"24"
"19":
  br label %"24"
"1a":
  br label %"24"
"1b":
  br label %"24"
"1c":
  br label %"24"
"1d":
  br label %"24"
"1e":
  br label %"24"
"1f":
  br label %"24"
"20":
  br label %"24"
"21":
  br label %"24"
"22":
  br label %"24"
"23":
  br label %"24"
"24":
  %".173" = load i64, i64* %"var_4h"
  %".174" = trunc i64 %".173" to i32
  store i32 %".174", i32* %"register0x0
```

```
    "
  %".176" = load i32, i32* %"
    register0x0"
  ret i32 %".176"
}
```