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

© Tejvinder Singh Toor 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 \hspace{1cm} ix

List of Figures \hspace{1cm} x

1 Introduction \hspace{1cm} 1

2 Background \hspace{1cm} 3
  2.1 LLVM \hspace{1cm} 3
  2.2 Ghidra \hspace{1cm} 4
    2.2.1 Low P-Code \hspace{1cm} 5
    2.2.2 High P-Code \hspace{1cm} 6
    2.2.3 Internal Decompilation Data Structures \hspace{1cm} 7
  2.3 Translating P-Code to LLVM IR \hspace{1cm} 7

3 Ghidra-to-LLVM \hspace{1cm} 9
  3.1 Overview \hspace{1cm} 9
  3.2 Disassembly \hspace{1cm} 10
    3.2.1 Disassemble Function Signature \hspace{1cm} 11
    3.2.2 Disassemble Instruction \hspace{1cm} 11
    3.2.3 Emit Register and Memory References \hspace{1cm} 12
  3.3 Lifting Stage \hspace{1cm} 13
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1 Lift Registers and Memory References</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2 Build Function and CFG</td>
<td>14</td>
</tr>
<tr>
<td>3.3.3 Populate Function and CFG</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Example of Preservation of Buffer Overflow</td>
<td>19</td>
</tr>
<tr>
<td>4 Ghidrall</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Decompilation</td>
<td>24</td>
</tr>
<tr>
<td>4.2.1 Call Graph Recovery</td>
<td>24</td>
</tr>
<tr>
<td>4.2.2 Function Decompilation</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Lifting Stage</td>
<td>26</td>
</tr>
<tr>
<td>4.3.1 Global Recovery</td>
<td>28</td>
</tr>
<tr>
<td>4.3.2 Calling Convention Recovery</td>
<td>28</td>
</tr>
<tr>
<td>4.3.3 Local Function Stack Recovery</td>
<td>28</td>
</tr>
<tr>
<td>4.3.4 Instruction Lifting</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Example</td>
<td>31</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>34</td>
</tr>
<tr>
<td>5.1 Simple Password Challenge</td>
<td>34</td>
</tr>
<tr>
<td>5.2 Functional Verification</td>
<td>36</td>
</tr>
<tr>
<td>5.2.1 Test Generation</td>
<td>36</td>
</tr>
<tr>
<td>5.2.2 Comparing Stack Structures</td>
<td>36</td>
</tr>
<tr>
<td>5.2.3 Comparing Lifters</td>
<td>38</td>
</tr>
<tr>
<td>6 Related Work</td>
<td>39</td>
</tr>
<tr>
<td>6.1 Lifters</td>
<td>39</td>
</tr>
<tr>
<td>6.2 Pharos</td>
<td>40</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>41</td>
</tr>
</tbody>
</table>
List of Tables

2.1.1 Examples of LLVM IR ........................................... 3
2.2.1 P-Code Operations Introduced at High P-Code Stage .............. 7
3.3.1 Mappings of P-Code to LLVM IR .................................. 18
4.3.1 Mappings of High P-Code to LLVM IR ............................ 30
5.2.1 Success Rate Overall for Structures ................................ 36
5.2.2 Comparison Local Function Stacks ................................. 38
5.2.3 Overall Success Rate of Lifters ................................... 38
List of Figures

2.2.1 Internal Representations of Ghidra Data Flow .......................... 5
2.2.2 Snippet of Low P-Code .................................................... 6
2.3.1 Translation of x86 Assembly to Low P-Code .......................... 8
2.3.2 Translation of Low P-Code to LLVM IR ................................. 8
3.1.1 Overview for Ghidra-to-LLVM. ........................................... 9
3.2.1 Comparison of source and intermediate XML Output .................. 12
3.2.2 Ghidra-to-LLVM XML instruction output ............................... 13
3.2.3 Example of XML output for register and memory references .......... 14
3.3.1 Comparison of intermediate XML and LLVM for registers and memory .. 16
3.3.2 Example of LLVM output of a function stack .......................... 17
3.4.1 Call Stack Setup ......................................................... 19
3.4.2 Example of a lifted LLVM IR in Ghidra-to-LLVM ....................... 20
3.4.3 Calling of strcpy from foo ............................................. 21
3.4.4 Calling strcpy ............................................................ 22
4.1.1 Overview for Ghidrall. .................................................... 23
4.2.1 Sample output of rizin afl .............................................. 26
4.3.1 Different Formats of Local Function Stack Recovery .................... 29
4.4.1 Comparison of source and Ghidra decompiled C ....................... 32
4.4.2 Ghidrall LLVM output for main ........................................ 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1 A Simple Password Challenge</td>
<td>34</td>
</tr>
<tr>
<td>5.1.2 Ghidrall and SeaHorn Instrumentation Functions</td>
<td>35</td>
</tr>
<tr>
<td>5.2.1 Test Generation</td>
<td>36</td>
</tr>
<tr>
<td>5.2.2 Example Functional Verification Problem</td>
<td>37</td>
</tr>
</tbody>
</table>
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 front-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.

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

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
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
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.
### 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 parameters, 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.

<table>
<thead>
<tr>
<th>High P-Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIEQUAL</td>
<td>Phi node for SSA</td>
</tr>
<tr>
<td>INDIRECT</td>
<td>Marker for decompiler for indirect change</td>
</tr>
<tr>
<td>PTRADD</td>
<td>Pointer addition</td>
</tr>
<tr>
<td>PTRSUB</td>
<td>Pointer subtraction</td>
</tr>
<tr>
<td>CAST</td>
<td>Type casting</td>
</tr>
</tbody>
</table>

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

1 foreach function do
2    DisassembleFunctionSignature(function);
3    foreach instruction do
4        DisassembleInstruction(instruction);
5    end
6 end
7 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, parameters, 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

<table>
<thead>
<tr>
<th>Result: Function Low P-Code XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Emit <code>function</code> name;</td>
</tr>
<tr>
<td>2 Emit <code>function</code> address;</td>
</tr>
<tr>
<td>3 Emit <code>function</code> output type;</td>
</tr>
<tr>
<td>4 <code>foreach</code> input do</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7 <code>end</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Result: Instruction Low P-Code XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Emit <code>assembly</code> address;</td>
</tr>
<tr>
<td>2 <code>foreach</code> P-Code op do</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8 <code>end</code></td>
</tr>
</tbody>
</table>

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
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
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

1 LiftRegistersandMemoryReferences;
2 foreach function do
3   BuildFunctionCFG(function);
4   PopulateFunctionCFG(function);
5 end

Algorithm 5: LiftRegistersMemory

Result: Register and Memory Data Structures in LLVM

1 foreach register do
2   if register is a flag then
3     Lift flag register;
4   else if register is a pointer then
5     Lift pointer register;
6   else
7     Lift generic register;
8   end
9 end
10 foreach memory do
11   Lift memory location;
12 end

Algorithm 6: BuildFunctionCFG

Result: Function Structure in LLVM

1 foreach function do
2   Build function type;
3   Build entry block;
4   foreach instruction address do
5     Build instruction block;
6 end
7 end
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

<table>
<thead>
<tr>
<th>Result: Function Population in LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

Function Stack

| 1 | entry: |
| 2 | "%stack" = alloca i8, i32 10485760 |
| 3 | "%stack_top" = getelementptr i8, i8* "%stack", i64 10485752 |
| 4 | store i8* "%stack_top", i8** @"RSP" |
| 5 | 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

<table>
<thead>
<tr>
<th>No.</th>
<th>P-Code</th>
<th>LLVM IR via llvmlite</th>
</tr>
</thead>
</table>
| 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   | CBRA \[ram\]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.

![Call Stack Setup](image)

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 0x00100674 (lines 1–3) and 0x0010067e (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 (0x00100688 – 0x0010069c) 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 "aaaaaaaaa" 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...
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 laid 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.
Figure 3.4.3: Calling of `strcpy` from `foo`
Figure 3.4.4: Calling `strcpy`
Chapter 4

Ghidrall

4.1 Overview

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
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. $to\_visit \leftarrow \{\text{entry}\}$;
2. $visited \leftarrow \{\}$;
3. while $to\_visit > 0$ do
   4. $current \leftarrow to\_visit.pop$;
   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. $to\_visit \leftarrow reference$
      14. end
      15. $visited \leftarrow current$;
   16. end
   17. foreach visited function do
      18. $DecompileFunction$;
   19. 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
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

1 Emit basic blocks in flat structure;
2 Emit return type reference;
3 Emit function name and address;
4 Emit parameter stack range;
5 foreach parameter do
6   Emit name, typeref, space and offset;
7 end
8 Emit local variable stack range;
9 foreach local variable do
10   Emit name, typeref, space and offset;
11 end
12 foreach basic block do
13   Emit id and address;
14   foreach operation do
15     Emit operation name;
16     Emit output varnode;
17     foreach input do
18       Emit input varnode;
19     end
20 end
21 Emit in and out branches;
22 end

Algorithm 10: Ghidrall Lifting Algorithm

Result: Valid Lifted LLVM

1 GlobalRecovery;
2 foreach function do
3   CallingConventionRecovery;
4   LocalFunctionStack;
5   InstructionLifting;
6 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 indepen-
dently 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 char-
acter 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 sup-
port 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.
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

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

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 ∗mystrcpy(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];
    mystrcpy((char ∗)c, bar);
}

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

(a) Source code

(b) Standard Ghidra Decompiler Output for main

Figure 4.4.1: Comparison of source and Ghidra decompiled C
define i64 0 main() {
  "0";
  %".2" = alloca %"local struct main",
  %var_40h" = getelementptr inbounds %"local struct main", %"local struct main"* %".2", i32 0, i32 0
  %uStack64" = getelementptr inbounds %"local struct main", %"local struct main"* %".2", i32 0, i32 2
  %uStack68" = 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
  %".18" = 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"
  store i32 %".37", i32* %uStack40
  %".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" = sext 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: Ghidra LLVM output for main

33
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".

<table>
<thead>
<tr>
<th>Ghidrall</th>
<th>SeaHorn</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT_RAND</td>
<td>nd()</td>
<td>Non-deterministic input</td>
</tr>
<tr>
<td>path_start()</td>
<td>-</td>
<td>Program start</td>
</tr>
<tr>
<td>path_goal()</td>
<td>verifier.error()</td>
<td>Program objective</td>
</tr>
<tr>
<td>path_non_goal()</td>
<td>verifier.error()</td>
<td>Program failure state</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Stack Format</th>
<th>Passes</th>
<th>Fails</th>
<th>Lifting Fails</th>
<th>Timeouts</th>
<th>Success Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_option</td>
<td>484</td>
<td>64</td>
<td>30</td>
<td>4</td>
<td>83.16</td>
</tr>
<tr>
<td>byte_addressable</td>
<td>483</td>
<td>87</td>
<td>8</td>
<td>4</td>
<td>82.99</td>
</tr>
<tr>
<td>single_struct</td>
<td>501</td>
<td>67</td>
<td>8</td>
<td>6</td>
<td>86.08</td>
</tr>
</tbody>
</table>

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
```c
#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.
### 5.2.3 Comparing Lifters

<table>
<thead>
<tr>
<th>Lifter</th>
<th>Passes</th>
<th>Fails</th>
<th>Lifting Fails</th>
<th>Timeouts</th>
<th>Success (%)</th>
<th>Avg. LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ghidrall</td>
<td>501</td>
<td>67</td>
<td>8</td>
<td>6</td>
<td>86.08</td>
<td>61.1</td>
</tr>
<tr>
<td>McSema</td>
<td>414</td>
<td>168</td>
<td>0</td>
<td>0</td>
<td>71.13</td>
<td>3417.6</td>
</tr>
</tbody>
</table>

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\cite{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\cite{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\cite{19}. The version of McSema that Ghidrall is tested against uses DynInst for the first stage\cite{20}. McSema has a few interesting features that Ghidrall does not. For instance, it handles C++ exceptions\cite{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 \texttt{invoke} instruction to call them. Ghidrall is currently unable to replicate this feature.

McToll\cite{14}\cite{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\cite{15}\cite{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


APPENDICES
Appendix A

Ghidra-to-LLVM’s bof.c

```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) {
    my_strcpy((char*)c, bar); // no bounds checking
}

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

46
Appendix B

Ghidra-to-LLVM's vuln.ll

1  ; ModuleID = lifted
2  target triple = x86_64-pc-linux-gnu
3  target datalayout = e-m:e-p:32:32-f64
4  :32:64-f80:32-n8:16:32-8128
5  
6  @CF = internal global i1 0
7  @RSP = internal global i8* null
8  @OF = internal global i1 0
9  @SF = internal global i1 0
10 @ZF = internal global i1 0
11 @PF = internal global i1 0
12 @RAX = internal global i64 0
13 @RIP = internal global i8* null
14 @EBP = internal global i8* null
15 @R9 = internal global i64 0
16 @RDX = internal global i64 0
17 @RSI = internal global i64 0
18 @R8 = internal global i64 0
19 @RCX = internal global i64 0
20 @RDI = internal global i64 0
21 @R13 = internal global i64 0
22 @R14 = internal global i64 0
23 @R15 = internal global i64 0
24 @R12 = internal global i64 0
25 @RBX = internal global i64 0
26 @R13D = internal global i32 0
27 @EDX = internal global i32 0
28 @EBX = internal global i32 0
29 @A_00300fe8:8 = internal global i64 0
30 @A_00100502:8 = internal global i64 0
31 @A_00300fc0:8 = internal global i64 0
32 @A_00300fc8:8 = internal global i64 0
33 @A_00300fd0:8 = internal global i64 0
34 @A_00100510:8 = internal global i64 0
35 @A_00300ff8:8 = internal global i64 0
36 @A_00100502:8 = internal global i64 0
37 @A_00100502:8 = internal global i64 0
38 @A_00100502:8 = internal global i64 0
39 @A_00300fd8:8 = internal global i64 0
40 @A_001005f0:8 = internal global i64 0
41 @A_00300ff0:8 = internal global i64 0
42 @A_00100510:8 = internal global i64 0
43 @A_001005f0:8 = internal global i64 0
44 @A_00100510:8 = internal global i64 0
45 @A_00300fd8:8 = internal global i64 0
46 @A_001005f0:8 = internal global i64 0
47 @A_001005f0:8 = internal global i64 0
48 @A_001005f0:8 = internal global i64 0
49 @A_001005f0:8 = internal global i64 0
50 @A_001005f0:8 = internal global i64 0
51 @A_001005f0:8 = internal global i64 0
52 @A_001005f0:8 = internal global i64 0
53 @A_001005f0:8 = internal global i64 0
54 @A_001005f0:8 = internal global i64 0
55 @A_001005f0:8 = internal global i64 0
56 define void @main()
57 {
58 0010066c:
59  % .20 = ptrtoint i8** @RBP to i64
60  % .21 = getelementptr i8*, i8** @RSP, i64 0, i64 -8
61  store i8* % .21, i8** @RSP
62 % .23 = getelementptr i8*, i8** @RSP, i64 0, i64 0
63 % .24 = bitcast i8* % .23 to i64*
64  store i64 % .24, i64* % .24
65  br label %0010066d
66 0010066d:
%.27 = getelementptr i8*, i8** @RBP, i64 0, i64 0
%.28 = bitcast i8* %.27 to i64
store i64 %.26, i64* %.28
br label %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 %.47, %.47
%.49 = trunc i64 %.48 to i1
store i1 %.49, i1* @PF
br label %00100674:
%
%.57 = getelementptr i8*, i8** @RBP, i64 0, i64 -64
%.58 = load i64, i64* @RAX
%.59 = bitcast i8* %.58 to i64
%.5a = load i64, i64* @RDX
%.5b = bitcast i8* %.5a to i64
%.5c = getelementptr i8*, i8** @RBP, i64 0, i64 -48
%.5d = load i64, i64* @RAX
%.5e = load i64, i64* @RDX
%.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 %001006 b8
001006 b8:
%.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 ()
{
0010064 a:
%.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 %.18 , i64 * @RIP
%.19 = load i8*, i8** @RSP
store i8* %.19 , i8** @RIP
%.20 = ptrtoint i8** @RSP to i64
%.21 = getelementptr i8*, i8** @RIP,
      i64 0, i64 -8
%.22 = bitcast i8* % .21 to i64 *
store i64 %.22 , i64 * @RAX
%.23 = load i8*, i8** @RBP
store i8* %.23 , i8** @RIP
%.24 = ptrtoint i8** @RSP to i64
%.25 = icmp ult i64 %.24 , 32
%.26 = call i64 @llvm.sadd.with.overflow.i64 ( i64 % .25 ,
      i64 32)
%.27 = extractvalue {i64 , i1} %.26 , 1
%.28 = call void @strncpy ( i64*, i64 , i1)
%.29 = extractvalue {i64 , i1} %.28 , 1
store i1 %.29 , i1* @CF
%.30 = getelementptr i8*, i8** @RSP,
      i64 0, i64 -32
%.31 = load i8*, i8** @RSP
store i8* %.31 , i8** @RSP
%.32 = icmp ult i64 %.31 , 32
%.33 = icmp ult i64 %.33 , 0
%.34 = icmp eq i64 %.34 , 0
%.35 = icmp eq i64 %.33 , 0
%.36 = load i8*, i8** @RSP
store i8* %.36 , i8** @RSP
%.37 = load i8*, i8** @RSP
store i8* %.37 , i8** @RSP
%.73 = getelementptr i8*, i8** @RSP, i64 0, i64 8
store i8* %.73, i8** @RSP
br label %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

1 ; ModuleID = "/tmp/examples/ password.bin"
2 target triple = "i386-pc-linux-gnu"
3 target datalayout = "e-m:e-i64:64-f80 :128-n8:16:32:64-S128"
4
5 %"local_struct.main" = type { i7999904 ,
i64}
6 declare i32 %"nd"()
7 declare void %"verifier.error"()
8 define void %"sym.path_goal"()
9 {
10 entry:
11 call void %"verifier.error"()
12 ret void
13 }
14
15 %"reloc.__libc_start_main" = global i32 0
16 %"sym.GNU_STACK" = global i32 0
17 %"obj.global_time" = global i32 0
18 %"reloc.time" = global i32 0
19 %"segment.LOAD1" = global i32 0
20 %"obj.__ctr" = global i32 0
21 %"reloc.__gmon_start" = global i32 0
22 define i32 %"main"()
23 {
24 %"iVar5" = alloca i32
25 %"iVar6" = alloca i32
26 %"iVar7" = alloca i32
27 %".2" = alloca %"local_struct.main"
28 %"padding" = getelementptr inbounds %
29 "local_struct.main", %"
30 local_struct.main"* %".2", i32 0,
i32 0
31 %"var_4h" = getelementptr inbounds %
32 "local_struct.main", %"local_struct.main"* %".2", i32 0,
i32 1
33 %"register0x8" = alloca i32
34 %"register0x206" = alloca i8
35 %"register0x0" = alloca i32
36 %".3" = call i32 %"nd"()
37 store i32 %".3", i32 * %"iVar1"
38 %".5" = call i32 %"nd"()
39 store i32 %".5", i32 * %"iVar2"
40 %".7" = call i32 %"nd"()
41 store i32 %".7", i32 * %"iVar3"
42 %".9" = call i32 %"nd"()
43 store i32 %".9", i32 * %"iVar4"
44 %".11" = call i32 %"nd"()
45 store i32 %".11", i32 * %"iVar5"
46 %".13" = call i32 %"nd"()
47 store i32 %".13", i32 * %"iVar6"
48 %".15" = call i32 %"nd"()
49 store i32 %".15", i32 * %"iVar7"
50 %".17" = load i32, i32 * %" iVar1"
51 %".18" = icmp slt i32 %".17", 97
52 br i1 %".18", label %"23", label %"1"
53 %"1":
54 %".20" = load i32, i32 * %" iVar1"
55 %".21" = icmp slt i32 %"122", %".20"
br i1 %".21", label %"23", label %"2" 117 %".65" = load i32, i32 * %"register0x8"

"2":

%.23" = load i32, i32 * %"iVar2" 118 %".66" = srem i32 %".65", 26
%.24" = icmp slt i32 %%.23", 97 119 %".67" = icmp eq i32 %%.66", 2
br i1 %.%.24", label %.22", label %.3" 120 %".68" = zext i1 %.67" to i8

"3":

store i8 %".68", i8* %"register0x206" 121 store i8 %".68", i8* %"register0x206"

%.26" = load i32, i32 * %"iVar2" 122 %".70" = load i8, i8* %"register0x206"
%.27" = icmp slt i32 %%.26", 97
br i1 %.%.27", label %.22", label %.4"

"4":

%.26" = load i32, i32 * %"iVar2" 130 %".77" = ashr i32 %%.76", 2
%.27" = icmp slt i32 122, %%.26"
br i1 %.%.27", label %.22", label %.4"

"5":

%.29" = load i32, i32 * %"iVar3" 137 %".84" = load i8, i8* %"register0x206"
%.30" = icmp slt i32 %%.29", 97
br i1 %.%.30", label %.21", label %.6"

"6":

%.32" = load i32, i32 * %"iVar3" 137 %".84" = load i8, i8* %"register0x206"
%.33" = icmp slt i32 %%.32", %%.32"
br i1 %.%.33", label %.21", label %.6"

"7":

%.35" = load i32, i32 * %"iVar4" 137 %".84" = load i8, i8* %"register0x206"
%.36" = icmp slt i32 %%.35", 97
br i1 %.%.36", label %.20", label %.7"

"8":

%.38" = load i32, i32 * %"iVar4" 137 %".84" = load i8, i8* %"register0x206"
%.39" = icmp slt i32 %%.38", %%.38"
br i1 %.%.39", label %.20", label %.8"

"9":

%.41" = load i32, i32 * %"iVar5" 144 %".90" = load i32, i32* %"register0x8"
%.42" = icmp slt i32 %%.41", 97
br i1 %.%.42", label %.10", label %.9"

"a":

%.44" = load i32, i32* %"iVar5"
%.45" = icmp slt i32 %%.44", 97
br i1 %.%.45", label %.1f", label %.a"

"b":

%.47" = load i32, i32* %"iVar6"
%.48" = icmp slt i32 %%.47", 97
br i1 %.%.48", label %.1e", label %.b"

"c":

%.50" = load i32, i32* %"iVar6"
%.51" = icmp slt i32 %%.50", 97
br i1 %.%.51", label %.1e", label %.c"

"d":

%.54" = load i32, i32* %"iVar7"
%.55" =icmp slt i32 %%.54", 97
br i1 %.%.55", label %.1f", label %.d"
162 %".107" = load i32, i32* %"register0x8"
163 %".108" = srem i32 %".107", 26
164 %".109" = icmp eq i32 %".108", 20
165 %".110" = zext i1 %".109" to i8
166 store i8 %".110", i8* %"register0x206"
167 %".112" = load i8, i8* %"register0x206"
168 %".113" = trunc i8 %".112" to i1
169 br i1 %".113", label %"12", label %"19"
170 %".115" = load i32, i32* %"iVar5"
171 %".116" = mul i32 %".115", 15
172 store i32 %".116", i32* %"register0x8"
173 %".118" = load i32, i32* %"register0x8"
174 %".119" = ashr i32 %".118", 2
175 %".122" = srem i32 %".121", 26
176 %".123" = icmp eq i32 %".122", 11
177 %".124" = zext i1 %".123" to i8
178 store i8 %".124", i8* %"register0x206"
179 %".126" = load i8, i8* %"register0x206"
180 %".127" = trunc i8 %".126" to i1
181 br i1 %".127", label %"11", label %"16"
182 %".129" = load i32, i32* %" iVar6"
183 %".130" = mul i32 %".129", 25
184 store i32 %".130", i32* %"register0x8"
185 %".132" = load i32, i32* %"register0x8"
186 %".133" = ashr i32 %".132", 2
187 %".134" = load i32, i32* %"register0x8"
188 %".135" = mul i32 %".134", 17
189 store i32 %".135", i32* %"register0x8"
190 %".136" = load i32, i32* %"register0x8"
191 %".137" = srem i32 %".136", 26
192 %".138" = icmp eq i32 %".137", 16
193 %".139" = zext i1 %".138" to i8
194 store i8 %".139", i8* %"register0x206"
195 %".140" = load i8, i8* %"register0x206"
196 %".141" = trunc i8 %".140" to i1
197 br i1 %".141", label %"14", label %"17"
198 %".143" = load i32, i32* %"iVar7"
199 %".144" = mul i32 %".143", 29
200 store i32 %".144", i32* %"register0x8"
201 %".146" = load i32, i32* %" register0x8"
202 %".147" = ashr i32 %".146", 2
203 store i32 %".147", i32* %"register0x8"
204 %".149" = load i32, i32* %" register0x8"
205 %".150" = srem i32 %".149", 26
206 store i32 %".150", i32* %"register0x206"
207 %".152" = zext i1 %".151" to i8
208 store i8 %".152", i8* %"register0x206"
209 %".155" = trunc i8 %".154" to i1
210 br i1 %".155", label %"15", label %"16"
211 %".157" = load i32, i32* %" iVar6"
212 %".158" = mul i32 %".157", 18
213 store i32 %".158", i32* %"register0x8"
214 %".160" = load i32, i32* %"register0x8"
215 %".161" = ashr i32 %".160", 2
216 call void @"sym.path_goal"()
250  %".176" = load i32, i32* %"
    register0x0"

251  ret i32 %".176"
252  }