TU WIEN Informatics

Embedded Binary Rewriting

Utilizing Ghidra and LLVM

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Raphael Ludwig, BSc
Matrikelnummer 01526280

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Wien, 1. Oktober 2021

Raphael Ludwig	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Embedded Binary Rewriting

Utilizing Ghidra and LLVM

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Raphael Ludwig, BSc
Registration Number 01526280

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Vienna, 1st October, 2021

Raphael Ludwig	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Raphael Ludwig, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Oktober 2021

Raphael Ludwig

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die mich bei der Erstellung dieser Masterarbeit unterstützt und motiviert haben.

Zuerst möchte ich mich bei meinem Betreuer, Georg Merzdovnik, bedanken, der mich mit hilfreichen Anregungen und konstruktiver Kritik bei der Erstellung dieser Arbeit unterstützt hat.

Ich möchte mich auch bei meinen Studienkollegen und Freunden bedanken, die mich mit Interesse und Hilfsbereitschaft unterstützt haben. Zahlreiche interessante Diskussionen und Ideen, haben dazu beigetragen, dieses Thema in der Masterarbeit zu behandeln.

Abschließend möchte ich mich bei meinen Eltern bedanken, die mir durch ihre Unterstützung das Studium ermöglicht haben.

Acknowledgements

At this point, I would like to thank all those who have supported and motivated me in the preparation of this master's thesis.

First of all, I would like to thank my supervisor, Georg Merzdovnik, who supported me with helpful suggestions and constructive criticism during the preparation of this thesis.

I would also like to thank my fellow students and friends who have supported me with interest and helpfulness. Numerous interesting discussions and ideas, have helped to address this topic in the master's thesis.

Finally, I would like to thank my parents, who have made my studies possible through their support.

Kurzfassung

Eine geeignete Strategie für das Patchen und Aktualisieren von Anwendungen ist ein wesentlicher Eckpfeiler einer modernen IT-Umgebung. Während in einem Open-Source-Kontext anfällige oder veraltete Systeme leicht gepatcht werden können, ist dies bei Closed-Source-Systemen nicht der Fall. Daher kann der Einsatz von Binär-Rewriting-Frameworks als vorteilhaft angesehen werden, insbesondere bei der Untersuchung von IoT-Anwendungen, da diese Anwendungen oft Closed-Source sind.

In dieser Arbeit wurde ein Prototyp eines Binär-Rewriting-Frameworks entwickelt, um die Möglichkeiten der Nutzung von Ghidra und des LLVM-Frameworks für den Umgang mit ELF-Binärdateien und eingebetteten System-Images für ARM-Prozessoren zu untersuchen. Die Abhängigkeit von einem binären Reverse-Engineering-Framework wie Ghidra kann als vorteilhaft für die Verarbeitung von Binärdateien und eingebetteten System-Images angesehen werden, da diese Plattformen bereits verschiedene Analysatoren für unterschiedliche Architekturen bereitstellen. Allerdings ist die Umwandlung der internen Repräsentation von Ghidra (P-Code) in soliden LLVM IR-Code nicht trivial, da nicht alle Sprachkonstrukte trivial aufeinander abgebildet werden können. Daher wird in dieser Arbeit die Transformation verschiedener Sprachkonstrukte wie Phi-Knoten, Typrepräsentationen und Zeigerarithmetik diskutiert, bevor wichtige Fallstricke aufgezeigt werden, die bei der Transformation von eingebetteten Systembildern auftreten können.

Darüber hinaus wurde der Prototyp an einigen ausgewählten Binärdateien evaluiert, um zu zeigen, dass der Transformationsprozess keinen nennenswerten Laufzeit-Overhead erzeugt. Die derzeitigen Einschränkungen des Prototyp- und Transformationsprozesses, wie z. B. der Umgang mit falsch identifizierten Codeabschnitten oder Datentypen und des Neukompilierungsprozess, werden kurz anhand der Abbilder der eingebetteten Systeme Zephyr und FreeRTOS aufgezeigt.

Abstract

A suitable strategy for patching and updating applications is an essential cornerstone of a modern IT environment. While in an open source context, vulnerable or outdated systems can be easily patched, this is not the case for closed source systems. Therefore, the use of binary rewriting frameworks can be seen as beneficial, especially when investigating IoT applications, as these applications are often closed-source.

In this work, a prototype binary rewriting framework was developed to explore the possibilities of using Ghidra and the LLVM framework to handle ELF binaries and embedded system images for ARM processors. The reliance on a binary reverse engineering framework such as Ghidra can be seen as beneficial for processing binaries and embedded system images, as these platforms already provide different analyzers for different architectures. However, transforming Ghidra's internal representation (P-code) into sound LLVM IR code is non-trivial, since not all language constructs can be trivially mapped to each other. Therefore, this thesis discusses the transformation of various language constructs such as phi-nodes, type representations, and pointer arithmetic before highlighting important pitfalls that can arise when transforming embedded system images.

Furthermore, the prototype was evaluated on a few selected binaries to highlight that the transformation process does not produce any noteworthy runtime overhead. The current limitations of the prototyping and transformation process, such as dealing with misidentified code sections or types and the build process, are briefly demonstrated using the images of the Zephyr and FreeRTOS embedded systems.

Contents

CHAPTER 1

Introduction

Binary rewriting can be a very powerful tool when it comes to patching and updating vulnerable or outdated systems. Not only allows binary rewriting the users to modify and harden applications, but also allows them to patch applications even when no source code is available. This especially helps in situations where closed source software is vulnerable and has to be patched in some way. Past attacks [1] have shown how vulnerable and dangerous a homogenous infrastructure can be when it is freely accessible on the Internet. This does not only apply to routers, but also affects IoT and edge devices that enable other resource constraint devices to connect to the internet and offload heavy computations into the cloud. Such IoT devices often utilize a real time operating system with a TCP/IP stack to achieve this. While security critical bugs [2] in such systems have serious consequences, not all devices might be patched by the manufacturer for various reasons. One reason could be that the manufacturer simply does not support the device anymore, which means that the end user has to buy a new device or life with a potential security problem. With an appropriate binary rewriter such problems can be eliminated without relying on the manufacturer and therefore can help to create more secure environments. But not only the firmware of the devices itself should be scrutinized, because IoT devices can often be controlled with a corresponding App. Which means that the security of the Smartphone and IoT devices are strongly linked together [3]. With an appropriate binary rewriter that focuses on firmware images of embedded systems and the ARM architecture, these problems can be managed to a certain extent and thus contribute to a safer environment. Nevertheless, there are many problems that are introduced when rewriting a binary without the source code. Because the binary has to be analyzed first, such a rewriting process is more complex than rewriting a program where the source code is available. Not only is it difficult to reconstruct the control flow from binaries, but also reconstructing the different data types that have been used throughout the program is not trivial [4].

1.1 Motivation & Problem statement

While the majority of binary rewriters focuses on the x86 instruction set, some binary rewriters for ARM also exist. But currently there are no IoT specific binary rewriters that can process firmware images. Although the ARM instruction set is not as complex as the x86 instruction set, most binary rewriters focus on the latter. Because the x86 instruction set uses variable length instructions, the binary rewriter has sometimes more options to transform the code, but decompiling and analyzing can be harder than with ARM. However, the ARM instruction set offers additional challenges that must be solved in order to rewrite binaries with minimal computing and space overhead. Embedded devices are often restricted, not only by processing power, but also by storage and memory, binary rewriters that target such platforms must take these limitations into consideration. Often the overhead that is introduced by binary rewriters renders most of the transformed binaries unusable for such resource constraint devices like a router or an IoT edge device. Because when rewriting a binary, the changes usually cannot be applied in place, which means that the resulting binary will naturally either grow or shrink, the binary rewriter must take this into account and also correct the relative and absolute jump addresses of the affected regions. In addition to direct manipulating of the assembly, there are also other ways to transform the binary by using an intermediate representation language in which the binary is lifted before analysis and can then also be compiled from it [5]. Although there are many such intermediate representation languages [6] used by software analyzing tools, not many binary rewriters utilize existing ones, but create their own internal ones for analyzing. The transformation process is then most of the time done directly at the assembly level without the help of a compiler that could try to reduce the overhead of such a transformation by applying optimizations.

1.2 Theses outline

The outline of the thesis is as follows. Chapter 2 will discuss background knowledge on various topics, such as the ARM architecture, Ghidra, and LLVM, to provide a brief introduction to topics that are relevant for binary rewriting. The next chapter 3 will cover the current state of the art of binary rewriters and highlight the differences between static and dynamic binary rewriters. Subsequently, chapter 4 discusses how to implement a transformation from P-Code to LLVM IR to create a binary rewriter that can export LLVM IR code using Ghidra. Not only is the transformation process of P-code operations and their structure discussed, but also how dynamically linked binaries or embedded system images can be handled in such a transformation. In chapter 5, this implementation is evaluated for both correctness and performance while discussing the discovered limitations of the developed prototype. Additionally, the process of using the developed prototype to patch a vulnerable program is highlighted. Finally, in chapter 6, the results of this thesis are summarized and further opportunities for research are discussed, as well as an outlook on various problems that were not addressed in this thesis.

CHAPTER 2

Background

This chapter focuses on highlighting various topics that will be needed throughout the thesis, such as the basics of ARM architecture and basic knowledge about disassembly. In addition, an overview of the technologies used in this work is given and important aspects of P-Code and LLVM IR are summarized.

2.1 ARM Architecture

Compared to the x86/x86-64 instruction set, which is present on processors that are mainly used for personal computers and servers, the ARM architecture is often found in mobile and embedded systems. To support not only phones or personal computers, but also low-power devices, ARM processors can implement various features that change the way these processors can handle data and code. For example, low power processors such as Cortex-M processors may only support the Thumb instruction set, while Cortex-A processors support the full AArch64 feature set and are capable of performing not only floating-point operations, but is also capable of performing simple cryptographic primitives such as hashing and en/decrypting data [7]. Such features are documented as a set of architectural profiles. For ARMv7 three such profile exist [7]:

- **ARMv7-A**: The ARMv7-A profile does support both Arm and Thumb instruction sets and requires the support of virtual addresses in the memory management model.
- **ARMv7-R**: ARMv7-R is the real-time profile, which also supports Arm and Thumb instruction sets, but does not require the support of virtual addresses.
- **ARMv7-M**: ARMv7-M is the profile for micro-controller and only has to support the Thumb instruction set. For this profile deterministic performance and a rather small size is important, as these processors are often used in low power devices.

Since the thesis focuses on embedded systems, further sections will only take the architectural profiles for micro-controller into account.

2.1.1 Registers

An ARM core contains a list of general purpose and special purpose registers. Unlike x86, where a full 32-bit register (EAX) can share its memory space with the respective smaller ones (AX, AL, AH), such a behavior is not possible in a register of an ARMv7-M processor. However, the AArch64 architecture has similar functionality. ARM processors have thirteen general purpose registers, labeled from r0 to r12. These registers have a fixed width of 32 bits. Besides these registers there are also some special purpose registers that fulfill a special role [7]:

- **SP**: The SP register, also known as Stack Pointer, is used to point to the stack of the program that is currently executed. Sometimes it is also named R13. Depending on which mode the processor is currently in, the register can represent the main stack pointer (MSP) or the process stack pointer (PSP).
- **LR**: The Link Register, which can also be referred as R14, is used to store the return address of a branching instruction. If this register is not used, it can also be used for other purposes.
- **PC**: The PC register contains the Program Counter of the application. The value of the register is the location of the current instruction plus 4 bytes.

In addition, there are various control and co-processor registers that can be used for floating point operations or other special operations that the ARM processor does not support. Depending on the processor used, it is also possible that other registers are banked in addition to the stack pointer, which means that the register is present multiple times on the processor and the current processor mode determines which version of the register is used [7].

2.1.2 Endianness

The Endianness of a processor specifies how data that is read from memory is interpreted. Most processors are either using big-endian or little-endian. As shown in Figure 2.1 the most significant byte is stored at the smallest memory address, while the least significant byte will be stored at the largest memory address. In little-endian the stored bytes are reversed, which means that the least significant byte will be stored at the smallest and the most significant byte will be stored at the largest memory address. ARM processors can support the functionality to switch from little-endian to big-endian when reading data bytes from the system memory. Instruction fetches, as well as access to control registers or the system control space are accessed with the little-endian memory system [7]. As shown in Figure 2.1, switching the endianness can result in errors when interpreting data that is

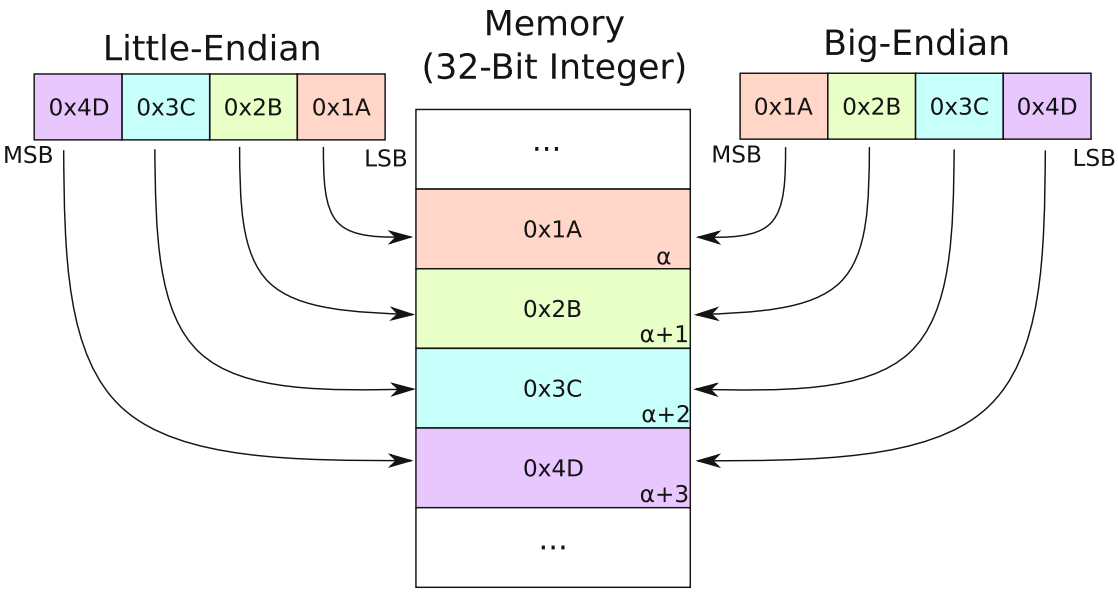

Figure 2.1: Endianness

located at an address α. In this example a single 32-bit integer is located at the address, which consists of 4 successive bytes (`0x1A, 0x2B, 0x3C, 0x4E`). When changing from little-endian to big-endian the value of this 32-bit integer changes completely and therefore it is important to take the endianness of the memory in account when accessing special memory regions, such as the system control space region.

2.1.3 Memory model

ARM processors use a rather relaxed memory model when compared to x86 processors. In this memory model hardware threads can execute load and store instructions out of order and therefore explicit synchronization mechanisms are needed to ensure consistency over all hardware threads when a single memory region is accessed. These synchronization mechanisms are data and instruction barriers, which ensure that no instruction after the barrier will be executed until all instructions before the barrier have been processed [8]. Besides a relaxed memory model, ARM processors do not restrict the type of stack that can be used by programs. It is possible to configure the type to be either full descending, full ascending, empty descending or empty ascending. But usually, full descending stacks will be generated by compilers. Although in most cases descending stacks are created, when working with embedded systems it is important to pay attention to the type of stack, since the direction of growth can not only vary, but also affect the placement of various memory segments when the binary is recompiled. Furthermore, memory that contains ARM or Thumb instructions must be aligned properly, otherwise the processor cannot correctly execute these instructions. ARM instructions are 32-bit aligned, while Thumb instructions are 16-Bit aligned [7].

2.2 Disassembly

During the disassembly of a binary file, the instructions are decoded and separated from the data in the binary file. For this, two different approaches can be used and even combined to achieve better results. The first approach is known as liner disassembly, which can be seen as simpler than the recursive disassembly approach, because the file will be scanned linearly from a starting position to an ending position and all data in between will be processed. Although, this approach is rather simple, it also has its disadvantages, because if a data region is encountered that can also be interpreted as valid instructions, the resulting disassembly may not be correct. The recursive approach however, is more complex, but does not suffer from this problem, because instructions are discovered by following the control flow of the functions that are analyzed. Which means that a precise control flow graph has to be created to fully discover all functions and blocks of instructions in a binary. In addition, the recursive approach has the disadvantage that if the destinations of a jump instruction are calculated at runtime, the destinations may not be determined at all [9].

2.2.1 ELF file format

The Executable and Linkable Format (ELF) file format, is a common standard for a cross-platform file format for programs and libraries that can be used by multiple operating systems. The ELF format specifies a header that includes memory segments, section headers and the data referred by these headers. Additionally, different endianness and multiple address sizes are supported to allow for flexible use [10]. When working with embedded system images, exporters are often able to export images to an ELF file format, which can then be further used in analysis tools such as Ghidra or in virtual machines (e.g. QEMU). These frameworks parse available headers and construct an environment for the program to be either analyzed or executed. For the binary analysis the sections with their respective permissions, as shown in Table 2.1, are important, because these can give hints to the framework about the intended usage of the contents. The permissions column lists the UNIX permission represent about the section, while the ELF-Type specifies the type of the section in the ELF file. SHT_NOBITS means, that there are no bytes associated with the section, while SHT_PROGBITS means that the ELF file provides the contents of the section.

Name	Permissions	ELF-Type
`.bss`	`rw-`	SHT_NOBITS
`.data`	`rw-`	SHT_PROGBITS
`.rodata`	`r--`	SHT_PROGBITS
`.text`	`r-x`	SHT_PROGBITS

Table 2.1: Excerpt of sections in the ELF file format [10]

2.2.2 Control Flow Graph

A control flow graph is a directed graph structure, as shown in Figure 2.2, that represents the control flow of functions. But also an inter-procedural control flow graph exists, which visualizes the control flow of functions rather then blocks of assembly code [11]. In a control flow graph the nodes are represented as blocks containing one or more assembly instructions, and the edges represent the destinations of direct or conditional jumps. To effectively use such a data structure for analysis, a block contains at most one branching instruction at the end of the block. Such blocks are then called basic blocks. Therefore, edges in Figure 2.2 can be colored differently depending on their type: blue (unconditional edge), green (true) or red (false) in case of a conditional jump. Such a graph cannot only be used to visualize the control flow structure, but also to group instructions together for further analysis, or be utilized by a binary rewriting prototype as described in Chapter 4.

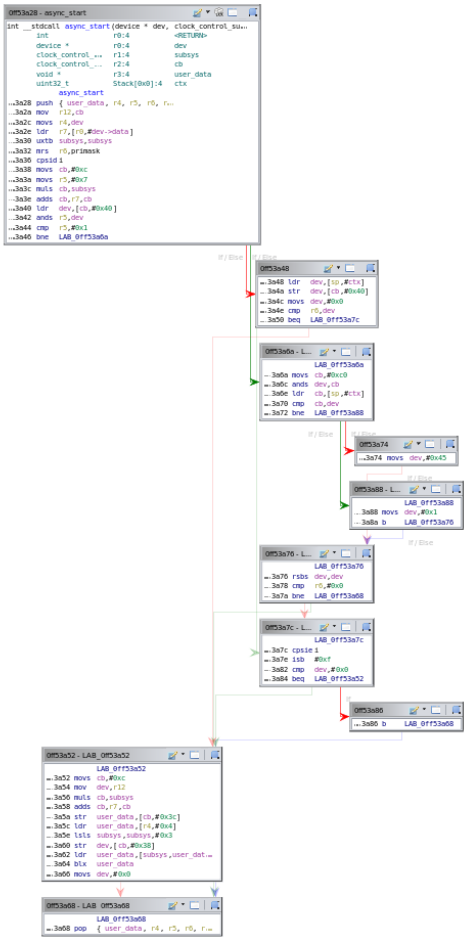

Figure 2.2: Ghidra: Control flow graph

2.3 Ghidra

Ghidra is an open-source software reverse engineering framework, which was published by the National Security Agency Research Directorate in March 2019 [12]. Besides the ability to disassemble binaries, Ghidra is also able to decompile and analyze binaries from multiple architectures and formats. With the newest release of Ghidra 10.0 [13], also an integration with a debugger is available to debug binaries directly in the reverse engineering framework. Because Ghidra features a plugin system, the functionality can be extended and new file format parsers as well as CPU targets or analyzing extensions can be easily added to help with the analysis of binaries. In order for Ghidra to provide a consistent analysis base for several different architectures, the disassembled binary program is converted to low-level P code on which the decompiler and other analyzers can operate to produce a high-level P code layer that can then be converted to pseudo-code, which is then displayed in Ghidra. This has the advantage that an algorithm for building the control flow graph, type detection, branch analysis, and other algorithms for binary analysis do not have to be implemented separately for each architecture, but only once for the low-level or high-level P-code representation. However, this also means that architecture-specific functionality and assembler instructions are abstracted or expressed differently by such a conversion.

2.3.1 SLEIGH

SLEIGH can be seen as a definition language that defines how machine instruction of a specific processor architecture should be mapped to P-Code. The language was derived from SLED and extended to be capable of encoding additional information about the control flow for the decompilation step and further analysis steps for Ghidra [14]. Since SLEIGH can be used for arbitrary architectures, basic restrictions and architecture specifics like endianness, alignment and registers have to be specified before any mapping from an instruction opcode to a P-Code operation can be provided. In order to be able to specify such basic architectural features in a modular way, SLEIGH implements a simple preprocessor language that is capable of including files and evaluating simple if expressions. Preprocessor constructs like `@ifdef` or `@ifndef` behave therefore very similarly to how `#ifdef` is handled in a C preprocessor. Simple instructions can be defined via the mnemonic, the parameters and different restrictions, such as the bitmask of the assembly instruction, as shown in Listing 1. In this Listing a simple unconditional branching instruction `b` is defined with `Addr24` being the parameter of the instruction, defining the target address of the jump. The rest of the definition are the conditions to ensure that the right bit pattern is matched for the instruction, which also includes to make sure that the processor is in ARM mode and the target address is defined. The body then includes the P-Code which should be generated for the detected assembly instruction [14].

```
:b Addr24 is $(AMODE) & cond = 14 & c2527 = 5 & L24 = 0 & Addr24
{
    goto Addr24;
}
```

Listing 1: Branching instruction definition in SLEIGH

2.3.2 P-Code

P-Code is a register transfer language, which features not only basic operations on integers and floats, but also operations that can impact the control flow. In Ghidra P-Code is used to represent the semantics of the underlying machine code as close as possible. All operations can have more than one input parameters, while either producing a return value or not. These input parameters or return values are named varnodes in Ghidra and cannot only represent the register or storage location, but can also contain more information from the decompiler and analyzer stages of Ghidra. Such information can be type and control flow information. While the type information is present as additional information in the high-level P-Code, it will not be present in the low-level P-Code that is displayed in Ghidra. The control flow information is provided by the decompiler only for all high-level P-Code operations in a graph structure containing all basic blocks of a function. Therefore, there is a distinction between low-level P-Code, which can be acquired by just lifting the assembly code into P-Code without any kind of analysis and the resulting high-level P-Code that is produced by the decompiler. This is important to keep in mind, because not only can some operations change during different levels of decompilation, such as `CALL`, `CALLIND` and `RETURN` but it is also possible that high-level P-Code operations, such as `BRANCH` or `CBRANCH` operations, inside a control flow structure can have a slightly other meaning than in their lov-level P-Code form. While a `CBRANCH` operation in low-level P-Code will set the program counter to a specific operation when a condition is met, the high-level version of this P-Code will actually make use of the control flow graph to represent such a change in the programs control flow by embedding this behavior into the true and false output edges of the respective block in the control flow graph. Besides these differences, there are also P-Code operations that are high-level specific operations and will not appear in low-level P-Code. Such operations can be `MULTIEQUAL` or `INDIRECT`, which have a special meaning, while not impacting the logic or the actual control flow of the program. A `MULTIEQUAL` operation is similar to a ϕ-expression in LLVM, while a `INDIRECT` operation only gives an indication to all analyzers working with high-level P-Code that a particular instruction in the binary code may also affect a particular input or output varnode of another P-Code operation [15].

Because there are far less P-Code operations then instructions in the ARM instruction set, one assembly instruction is most likely translated into multiple P-Code operations. Besides basic control flow operations for branching and calling functions, there are also only basic arithmetic operations and comparisons for integer or floating-point numbers

of variable length. For example, the integer addition operation `INT_ADD` can operate on any sized input, with the only the restriction that both inputs must have the same size. Although any input and output varnode of a P-Code operation can be either a register, or a variable in local or global scope, a varnode can also encode constants and different address spaces, which can be useful for embedded devices, where data and executable storage are separated entities [15].

2.3.3 Varnode

Varnodes are used as input and output variables by P-Code operations and can therefore be considered as unit of data that these operations process and produce. Such a varnode cannot only represent a fixed constant value, but also reference variables, registers or even memory regions that span over multiple addresses if necessary. Varnodes themselves do not have a specific type, as this information is missing in low-level P-Code, but can be enhanced with higher-level types and references to symbols in high-level P-Code that can then be displayed in the decompiled C-style code in Ghidra. Because a varnode is universally usable, a lot of information is stored in varnodes, the most important properties for the binary rewriter are listed below:

- **Address range**: A varnode does contain at least a start address, at which the data is stored. Such an address can point to various memory spaces, such as register or normal nonvolatile memory. While a register restricts the maximum size of the varnode, this is not necessarily true, if the varnode is stored somewhere in memory. In such cases, the annotated type or the P-Code operations used are used to limit the size of the varnode. The end address is never saved into a varnode and has to be calculated from the length and the start address.

- **Length of the data**: As already mentioned, the length of the data in each varnode can vary depending on the usage and the contained data. In most cases the length of the varnode matches the contained data type, but this may not necessarily be true for complex data types, such as structures, because they are often split to multiple varnodes. This is necessary since not all P-Code operations can handle these data types.

- **High-Level Information**: The amount of high-level information, such as the presence of a data type, a high-level variable, or even a symbol, depends on the level at which the P-code operations were generated. In a low-level analysis these types are missing, while in a high-level analysis most varnodes contain a higher-level data type or symbol. A missing data type cannot only indicate errors in the analysis, but also be an indicator for code that might have been obfuscated.

- **Definition**: A varnode also caries information about its definition, where the varnode was previously modified or assigned a new value. This most of the time refers to another P-Code operation where the varnode was used in the output slot,

but this can also be undefined if there is no logical definition of the varnode. This is often case if a varnode is a parameter of a function.

- **References of usage** Besides these properties, a varnode can also reference all P-Code operations where the varnode will be used as parameter. This can be used to analyze the flow of the data throughout the function.

2.3.4 Address spaces

According to the usage, all addresses of Varnodes are assigned to address spaces in Ghidra. Such address spaces can be the normal nonvolatile memory (RAM), the stack memory region or even a constant memory space,which can be an indicator that a certain varnode can be considered a constant value and never changes throughout the execution of the program. In P-Code, there is also a unique address space that is used to mark varnodes that are not overwritten by a P-Code operation throughout their lifetime, but can have the same physical memory address as other varnodes. These address spaces can be used in the analysis step to perform different kinds of optimization and help in identifying different variables and how the data is moved within a function. Such address spaces can also be used to mirror hardware specific memory spaces, such as the separation of ROM, flash and RAM. But this is usually not the case when a binary is analyzed by Ghidra, because hardware specific information would be needed for the binary analysis step to build such address spaces.

2.4 LLVM

LLVM is a powerful compiler infrastructure that does not only support multiple different backends, but also different frontends. One such fronted is for example clang, which can translate C/C++ code into the LLVM intermediate representation (IR) that is used by the compiler infrastructure to not only optimize the code, but also by the compiler backend to produce machine code for the target architecture [16]. The LLVM IR is s static single assignment (SSA) language, which is also strictly typed to ensure type safety when compiling any code into assembly instructions. With this intermediate representation it is possible to represent all high-level languages without any modifications to the language specification. This is possible because the LLVM IR provides common low-level operations that can be used to build higher-level constructions. But this also means that for some languages a transformation to this intermediate representation can be more difficult than for others [17].

While there are multiple representations of the LLVM IR, such as in-memory, on-disk bitcode (.bc) and a human readable language representation (.ll), this section will focus on the latter two representations as they are used latter on by the binary rewriting framework. Additionally, any LLVM IR code that is generated with the LLVM framework can be easily serialized into both of these representations and therefor also easily inspected and exchanged between different programs. Although the language syntax allows certain

constructs, as shown in Listing 2, these expressions will later then be rejected by a verification pass, as all expressions in the LLVM IR must be well formed. In the shown example the SSA property is violated and therefore it will result in a parser error when trying to compile such code with the LLVM framework [17].

```
%result = add i32 %result, 1
```

Listing 2: Non well formed LLVM IR expression

2.4.1 Static Single Assignment (SSA)

As already mentioned, the LLVM IR is a static single assignment from, which can be seen as a restrictive property of the intermediate representation [18]. Because of this property a variable must only be assigned once and cannot be used before it was defined, which makes the example shown in Listing 2 not well formed. While such a property can be limiting when transforming other languages into the LLVM IR, a SSA language can then be used to identify unused variables and can also help in allocating registers for different variables. Although a static single assignment form can easily be used for analysis and optimizations, representing local variables in a stack frame or referencing other variables from other blocks can be difficult. Therefore, LLVM IR allows for allocation of stack memory with **`alloca`**. While this expression yields a pointer to the allocated memory, the pointer has a type information attached to it, which must be specified in the **`alloca`** expression. With the help of this pointer local variables that reside on the stack can be accessed. Although referencing a single variable in another block is easily possible in a SSA language, it can get problematic in case of loops or other constructs that expect a value from the previous executed block is present. For this reason, ϕ-expressions (nodes) are needed [17].

2.4.2 ϕ-expression

In a SSA language ϕ-expression (nodes) are mainly used to solve a fundamental problem when working with such representations. A variable can only be assigned once and afterwards it is not possible to change the variable in any way. Therefore, when working with loops, or pieces of code that can have multiple incoming edges in one code block, a problem arises. Namely how should these changes be propagated and how can these variables be referenced correctly. In LLVM IR such ϕ-expressions must dominate all other expressions in a block and have to have a value for each incoming edge of the current block in the control flow graph. As shown in Figure 3 the variable `result` will reference a different value depending on the previous block. Such ϕ-expression can therefore also be used to easily represent loop counters in a SSA form, because referencing, both the initial value and the value from the previous iteration is possible without reassigning a new value to the variable [17].

```
%result = phi i32 [ 0, %entry ], [ %next, %block1 ]
```

Listing 3: Example ϕ-expression

2.4.3 Types

The LLVM internal representation language is a strongly statically typed language. In LLVM IR both, the declaration of a function or variable and the point of usage an appropriate type has to be specified, which will then be enforced throughout the program. In case of any type mismatches, a compilation will fail, as there are no automatic casts from on type to another, even when they would be compatible with each other. In LLVM IR variable sized integers and a number of common float representations can be considered as primitive types. When an integer type is used, the width of the integer has to be specified beforehand in the following format: `i<width>`, where width is the size of bits the integer should have. For example, when defining a 4-byte integer, the `i32` type is used. This can be useful for handling various high-level types that either have a target specific or custom size. One such high-level type can be a boolean, which can be converted into an integer with width 1 bit (`i1`). Besides these also pointers, arrays and structures can be created in the type system [17].

2.4.4 Global symbols

Global symbols in LLVM IR can be global variables or functions that are visible in the whole compilation unit. These symbols always have to start with an `@` character and can be seen as a pointer to their storage space. In LLVM IR this is needed, because a global variable describes most of the time a region in memory that can be accessed by any function in the compilation unit. Since every memory region in LLVM IR is accessed by a pointer, a global variable should also reflect this idea. Additionally, LLVM IR is also in a value SSA form and therefore changing a value without pointers can be a complicated task. Therefore, to accessing the contents of such global symbols often requires the usage of **`load`** or **`store`**, or **`call`** expressions. Global variables can optionally specify different properties, such as the linkage type and properties such as `unnamed_addr` or `local_unnamed_addr`, which can be used by the compiler to optimize the behavior of a global variable further. For example, if `unnamed_addr` is given, the compiler may merge constant values with other constant values that have similar contents, as the property signals the compiler that the address of this symbols is not relevant for the program. Global variables can also contain a property that defines the section name in which the global variable should be contained in the final binary [17].

2.4.5 Functions

Functions in LLVM IR can be seen as special global symbols that point to a region of executable memory. A function in LLVM IR has to be at least declared and an external property has to be added if the function is not contained in the current compilation unit. Otherwise, a function needs an entry block, which then contains all LLVM IR expression

that should be contained in the function. Since the name of these blocks is only local to the function, multiple functions can have the same block labels without interfering with each other. A function in LLVM IR, must always have a return value, if the function should not return anything the special value **`void`** as to be chosen, which also signals the calling site that the respective **`call`** expression does not have a result that can be assigned to a variable [17].

2.4.6 Intrinsic Functions

When generating LLVM IR expressions from P-Code operations or even directly assembly code, it is possible that not all of these operations can be expressed in expressions that are available in the LLVM IR language. This can be a common case when working with code that is either highly optimized for an architecture, or a lot of `__builtin_` functions have been used by the developers or statically linked libraries in the application. One way to handle such occurrences is to rely on LLVM intrinsic functions. Most of these functions that can be applied in such cases are all platform specific and therefore produce non-portable code and should not be used directly if not really necessary. Although, most of the time intrinsic functions simply represent platform specific instructions such as `strex` or `ldrex` also other more portable intrinsic functions exist. One example for such a function would be `llvm.read_register.i32(metadata)`, which allows at least in theory to read the contents of a register. Nevertheless, it can only be used to retrieve the stack pointer on certain platforms. The same restrictions can also be applied to the `llvm.write_register.i32(metadata, i32)` function, which is able to write a value to a register. In the intermediate representation such intrinsic functions behave exactly like any other normal functions and therefore also have to be declared with the right signature before they can be used [19] [17].

2.4.7 Inline Assembly

Not all generated assembly or P-Code constructs can be represented as normal expressions or intrinsic functions in LLVM. This can be the case for highly optimized or customized code, which cannot only be found in user space applications, but also in kernel space. Especially when working with embedded images this can be the case, because no library can be used to abstract interactions between user space and kernel space, or other processes such as handling interrupts and switching tasks or accessing hardware or platform specific registers. Such interactions usually rely on hand crafted assembly code, which is often hidden in platform specific frameworks that provide the functionality. If such constructs are encountered while the conversion between P-Code and LLVM is in process, these instructions have to be converted into inline assembly, otherwise these instructions would be missing from the final export and can therefore lead to an incomplete and wrong application. Compared to the C/C++ interface of the LLVM toolchain, the usage of inline assembly is rather limited in the LLVM IR language, because all inline assembly code has to be encoded as a function call [17]. While this still allows the usage of arbitrary assembly instructions in a controlled manner, it also

ensures through the semantic of the function call that these variables can be freely used as input parameters for an assembly instruction and that the compiler can also retrieve the return value from a register.

```
%1 = call i32 asm sideeffect "mrs $0, BASEPRI", "=r"()
```

Listing 4: Example LLVM IR inline assembly

An example for an inline assembly function call can be seen in Listing 4. The example encodes an access to the MRS register to retrieve the base priority mask and can often be found in kernel-space code. When creating such inline assembly function calls, the keyword **sideeffect** is often used to ensure that the compiler knows that side effects can occur that are not apparent through the constraint list. This also prevents the compiler from removing the assembly in a optimization stage. In this case the LLVM IR framework does not provide any intrinsic functions for such a platform specific behavior, but since the conversion is rather trivial the binary rewriting framework can either choose to rely on other third-party library that implement such a behavior or supply a direct inline assembly call as shown in Listing 4

CHAPTER 3

State of the art

This chapter gives an overview of the current state of the art by summarizing important aspects of binary rewriters and highlights different approaches. Since binary rewriters must disassemble at least parts of the binary to be able to modify it, the quality of the used disassembler is very important to such rewriters. Not only dictates the disassembler which binaries the rewriter can process, but the rewriter also has to take into account that the process of disassembling a binary is not a straightforward process and depending on the compiler optimizations the disassembled binary might differ from the actual source code that should be patched accordingly [5]. Most binary rewriters that currently exist, can be grouped into the following categories: static, dynamic and hybrid. These categories not only describe the basic approach that a rewriter follows, but also makes comparing different binary rewriters with each other easier, because rewriters from different categories might choose different approaches, such that the process might not be applicable for the same problem. Although most static binary rewriters rely on a control flow graph, which is typically extracted by a decompiler, to rewrite the binary, E9Patch [20] tries to accomplish this transformation by only using control flow agnostic methods. But because these methods are based on heuristics E9Patch does not have a 100 percent coverage and the overhead that is introduced makes it not suitable for resource constraint devices. Additionally, static rewriting or patching is not always possible for some resource constrained devices that provide real-time services, a patching mechanism can take advantage of the hardware-specific characteristics of such devices to inject code that can patch the system. On embedded systems that are powered by ARM SoCs this can be done by utilizing the hardware debugging unit, which cannot only load and rewrite the current program, but can also change the control flow via hardware breakpoints [21]. Besides static and dynamic binary rewriters, which will be discusses further in Section 3.2 and Section 3.3, also hybrid binary rewriters exist. These types of rewriters utilize both rewriting methods when rewriting binaries to improve their overall success when handling complex binaries.

3.1 Basics of binary rewriting

Regardless of the approach used by a binary rewriter, the key steps that such a rewriter must perform can be summarized as follows [5] [22]:

- **Parsing**: Executable binaries can have different file formats, such as ELF, PE or special formats for firmware images that are specifically crafted for a micro controller, which have to be parsed by a binary rewriter in order to access the contained information. While data that is contained in executable file formats, such as ELF and PE, can be grouped into two different data parts, namely administrative and payload, this is not necessarily true for firmware images that have been copied from en embedded system. The administrative data part usually provides further information about different sections that are contained in the binary. For ELF binaries the most important sections for the analysis step have been discussed in Subsection 2.2.1. Such administrative sections can also contain debug or type information, which can be used to help to recover more code more accurately in the analysis step. On the other hand, the contents of the payload section are often raw byte streams that can contain not only instructions, but also initialization data for global variables. If no debug information is attached to such a byte stream, the stream should be considered completely untyped, as no information is present on the location and the size of global variables or functions.

- **Analysis**: The purpose of the analysis step is to process the byte streams of instruction data from the previous step and recover as much of the programs control flow as possible. Depending on the strategy that is used by the binary rewriter, a simple disassembler might be sufficient, because a fully recovered control flow graph is not needed in the next steps. Instruction punning [23] can be seen as such an approach that would not need a fully recovered control flow graph to modify the binary, because in such a transformation a jump instruction is inserted in place of another instruction, which will then be copied over to the jump target with the code that should be executed. But if such a control flow graph, type or structural information is needed, then the use of a proper decompiler may be beneficial, otherwise a binary rewriter would have to implement such algorithms themselves. Recovering this type of information is not trivial, as the raw byte streams of instructions are completely agnostic to high-level constructs, such as data types, structures, unions or functions. A decompiler has to recover these from the semantics of the underlying instructions and then use the information to reconstruct as much useful higher-level code as possible.

- **Transformation**: After the binary has been analyzed, such that enough information for the transformation process is available, the transformation step can begin. In this step the binary rewriter starts to make changes to the control flow. Depending in the approach of the binary rewriter, these changes can be done by rewriting an instrumentation point, by changing a series of instructions in a

basic block or even by utilizing breakpoints, if the target system has support for such a feature [21]. Depending on how fine-grained a binary rewriter allows for a transformation to happen, other approaches have to be chosen, but it has to be minded that the more fine-grained a transformation process is, the more overhead will be encountered in the overall process.

- **Code generation**: In the code generation step, the binary rewriter persists the changes in an executable form, the result of the code generation step is not necessarily the same for all different types of binary rewriters. For static binary rewriters such a form would be an executable binary in an appropriate file format. In most cases these formats will be ELF or PE files. Since an embedded systems utilize custom formats these formats have to be converted to another representation if the binary rewriter has no support for an appropriate file format that can be used by such an embedded system. To persist these changes, a binary rewriter could place the newly generated code into additional sections that contain all the transformed code, or utilize a memory patching algorithm to change the control flow direly after the binary has been loaded by the operating system. While the second approach leaves the original untouched an additional loader is needed for the changes to be applied. But also utilizing existing compilers to create a completely new binary is a possibility. Dynamic binary rewriters on the other hand, may not have the ability to persist the output of the transformation step, but therefore can use the generated code directly at runtime and execute it.

3.2 Dynamic binary rewriter

Dynamic binary rewriters manipulate the binary while it is executing and take advantage of the runtime information of the process. How such a dynamic binary rewriter can work is pictured in Figure 3.1. When the rewriting process is started, the binary is loaded into the memory, where the instructions can be transformed after the initial analysis step. There are multiple approaches how a dynamic binary rewriter can achieve this goal. For example, all instructions can be interpreted by the rewriting engine, which can lead to a higher overhead then with other methods. One other method would be to use breakpoints at locations where transformation has to take place or the control flow of the application has to be changed. For this approach a dynamic binary rewriter can try to attach itself in a similar fashion to the target binary like a debugger [5]. Depending on the dynamic binary rewriter, such an architecture can also be more sophisticated than in other cases. For example, Instrew [24] uses a client-server architecture to split the actual execution and code caching from the parsing, transformation and optimization. While this approach can be used in a very flexible manner, it can also cause noticeable runtime overhead, because all functions that are identified by the client process and are getting transformed by the server are transferred as ELF objects from one side to another. Since the rewriting process is performed by the server process, the client process has to cache received code in form of a code cache, which can speed up the execution time, but

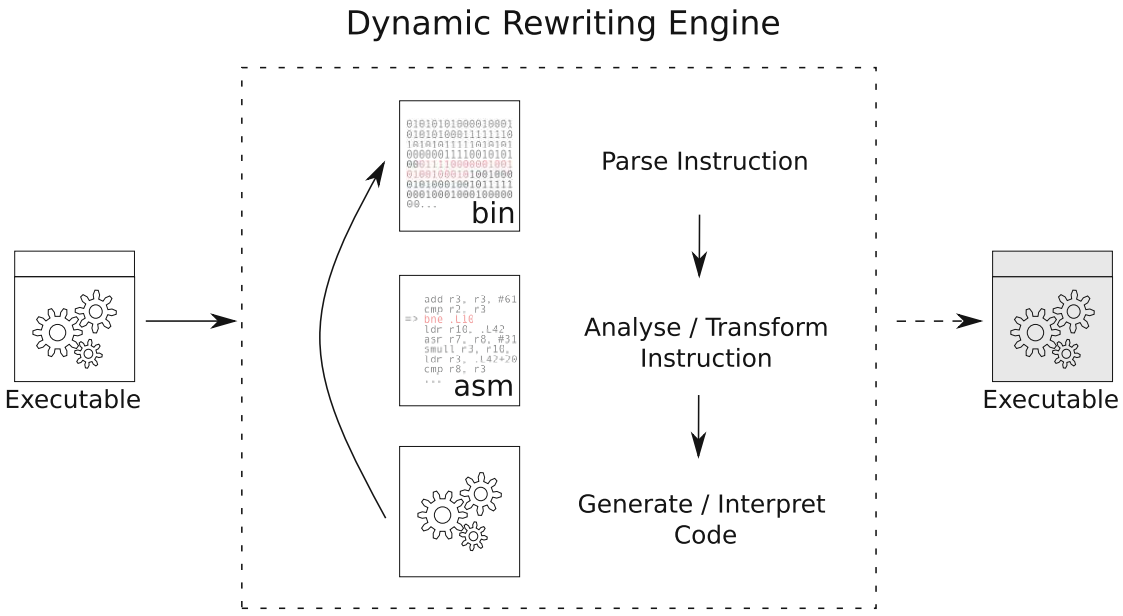

Figure 3.1: Visualization of a dynamic binary rewriter process

requires a special execution manager that is able to prevent the program from executing unknown pieces of code [24]. Although this approach limits the analysis of the binary to the execution paths that it takes during the runtime, binaries, which try to obfuscate their code by utilizing unpacker and encrypted memory segments can be processed by a dynamic binary rewriter. BinRec [25] is also build with such a principle in mind, because the whole binary rewriting process is build upon the analysis of execution path that are taken by simulated inputs. But while not all inputs can reliably be recorded and relayed to the binary, such as network requests to live services, the rewriting process accounts for such missing inputs by providing incremental rewriting support. While a dynamic rewriting approach can handle such situations better than a static binary rewriter, it comes at the cost of a computing and memory overhead, as the analysis process and the patching process are performed at runtime [5]. Instrew and BinRec use the LLVM IR as intermediate representation for modifications before the binary gets compiled, other dynamic binary rewriter such as HERA [21] have a different approach. Since HERA is a specialized dynamic binary rewriter for embedded real time systems it has to take into account that the important scheduled tasks can be completed in real time. To allow for this, HERA uses the idle time of the processor to copy the patched code into a new memory region the hardware debugger functionality to install hardware breakpoints, such that the switch from the non-patched version can happen atomically. Instead of halting the execution of the task, HERA uses the debug functionality to automatically redirect the control flow to the patched code segment. A dynamic binary rewriter does not necessarily produce a patched binary that can be used after the rewriting process without the dynamic binary rewriter engine.

3.3 Static binary rewriter

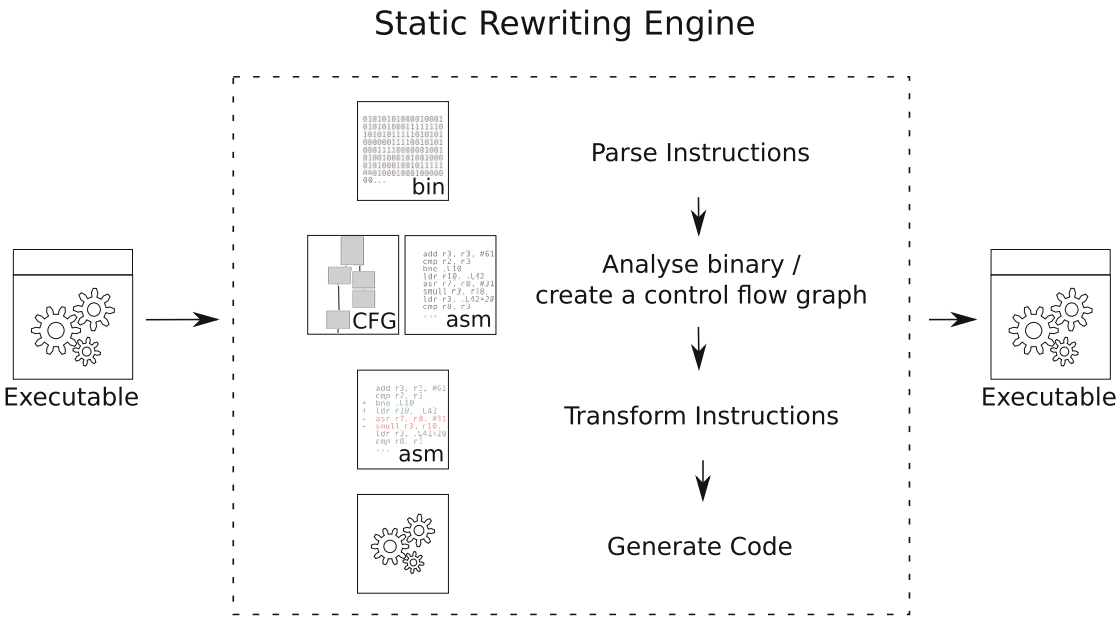

Figure 3.2: Visualization of a static binary rewriter process

In contrast to dynamic binary rewriter, static binary rewriter processes the binary by using static analysis, which means that the binary is most likely be processed as a while in each step before a patched output binary is produced, as shown in Figure 3.2. Static binary rewriters for ARM such as RevARM [26] or REPICA [27] use insertion-based approaches. In such an approach the code that is rewritten is inserted in the binary at the original location. This also means that if the rewritten code is bigger than the old one, all following direct and indirect jumps must be patched accordingly. While RevARM uses IDA Pro[1] for disassembling and creating the control flow graph, REPICA only uses the capstone[2] disassembler for preprocessing the binary. Although RevARM has a lower runtime overhead then other binary rewriters, dynamically generated, obfuscated or position independent code cannot be handled. Compared to other binary rewriters the overall space overhead that is introduced by the transformation of REPICA can be seen as negligible. But it does not seem to be suitable for firmware images for embedded systems, since it only focuses position independent android binaries and transforming other kinds of binaries might have undesired side effects. It has also been shown that inserting a shadow stack into the binary with REPICA can be done with less overhead than with Multiverse [27][28].

Instead of relying on a framework specific intermediate representation language, frameworks such as McSema [29] or Bin2llvm [30] lift the binaries directly to LLVM IR, which

[1] https://www.hex-rays.com/products/ida/
[2] https://www.capstone-engine.org/

then can be used to rewrite the binaries. But since lifting a binary into the LLVM IR is a very complex task, not all binaries can be processed since the lifting process is also not perfect and suffers from similar problems as it is the case with normal decompilers. Additionally, since the LLVM IR was built to support only compiler optimizations, an analysis of the binary in such a language can also be more troublesome than in an intermediate representation language that was built for such a task. But despite those shortcomings the binary rewriting process, fuzzing or even searching for vulnerabilities in such frameworks can be done in the LLVM IR [31]. This also eliminates the need to manually correct direct and indirect jumps, since the resulting code will be compiled by a compiler. While most modern binary analysis tools like IDA, Ghidra[3] or radare2[4] use an independent language to perform code analysis, compared to LLVM IR these other representations are only used for analysis and it is not supported that changes can be written back to assembly code [6]. Nevertheless, some solutions like RetroWrite [32] and SN4KE [33] utilize the features of such an intermediate representation language to perform instrumentation, fuzzing as well as mutation for binaries. SN4KE uses the GTIRB format, which is a high-level container that can store the structure of the binary in a format that can be processed by analysis and binary rewriting software. Additionally, to this feature, the original assembly is preserved, which makes it easier for insertion-based approaches to transform the binary while still being able to perform analysis of the control flow graph. But while GTIRB tries to encapsulate these important aspects for binary rewriters, a transformation into LLVM IR is still not trivially possible, because individual instruction still has to be transformed [34].

3.4 LLVM IR binary rewriter

LLVM IR binary rewriter are dynamic or static binary rewriter that utilize the LLVM intermediate representation in their rewriting process and therefore must have the ability to transform the disassembled or decompiled program into this intermediate representation before it can be modified. This section will briefly describe how Instrew and McSema handle this process.

3.4.1 Instrew

Since Instrew [24] is a dynamic binary rewriter, the binary is modified while it is being run, which means that the overall process of lifting, transforming and executing the newly generated code must be rather fast, otherwise the runtime performance will suffer from utilizing the framework. Besides basic modifications, also instrumentation is supported by Instrew and a client/server model is used in the transformation process. The server will be tasked to lift the assembly code, transform the lifted instructions to LLVM IR, apply the desired patches and compile back, such that the client can execute the newly generated code block. In this process the server process will first decode all instructions in a block,

[3] https://ghidra-sre.org/
[4] https://rada.re/n/

which means that from the current position of the program counter all instructions will be scanned linearly until a branching instruction is encountered, before attempting to transform the instructions. Then these decoded instructions are transformed to LLVM IR and if necessary ϕ-expressions are generated before the generated code can be instrumented, changed or compiled. After the compilation, the compiled code segment is sent to the client progress, which starts to execute the code until a missing entry in the code cache is encountered.

3.4.2 McSema

When a binary is rewritten with McSema, the binary is processed in two stages. In the first stage the binary will be analyzed and disassembled, which is done with the help of the Remill framework[5] and IDA Pro. This will generate a control flow graph with additional information for the next step. In this step the instructions will be lifted with the provided control flow into LLVM IR code. Compared to other binary rewriter, the transformation process of McSema uses a semantical approach where a series of instructions or even individual instructions are lifted as functions into the LLVM IR code. The implementation of these function can then vary depending on the target architecture and are often inlined in the compilation process to prevent the generation of additional overhead of a function call. Besides this the McSema binary rewriter will also create structures for handling the overall CPU and memory state of the application [29].

[5] https://github.com/lifting-bits/remill

CHAPTER 4

Implementation

This chapter describes a rewriting approach that can be used to convert Ghidra's high-level P-Code representation into LLVM IR. Figure 4.1 gives an overview of an example architecture of such a Ghidra plugin. The plugin does not only need the appropriate binaries as an input, but also the patchset and additional options, which can be used to customize the rewriting process. Such options are for example the different approaches to handle the linking of dynamically build binaries, as described in Section 4.9. While Ghidra provides a variety of different types of analyzers and binary parsers, a combination of them with different settings will produce different results and can impact the quality of the resulting transformation. But since the Ghidra plugin is encapsulated into a single action, also third-party analyzers can be used in Ghidra to improve the high-level P-Code representation. The resulting P-Code of the analyzing step is then used by the transformation step to convert the P-Code operations to LLVM IR expressions. Several issues must be addressed in this conversion, such as creating appropriate type information, handling stack variables, accounting for external dependencies, and more. This is because when P-Code operations are converted to LLVM IR expressions, not all P-code operations result in a single expression. Depending on the operations it can be zero or even multiple non-trival expressions. For example, a single `COPY` operation in P-Code copies the value from the input variable into the output variable. Such an operation can have multiple meanings in the LLVM IR environment, because depending on the used input other expressions as just an assignment operation may be needed. This is mostly the case, because P-Code does not distinguish between local or global symbols in the same way it is the case in LLVM. Although Ghidra represents global symbols as data labels and LLVM IR represents them as global pointers, many issues can arise when transforming different operations, which is discussed in more detail in Section 4.5. But also working with pointers and constant values can be a problem when transforming a binary, as absolute positions will most likely change during code generation, therefore additional measures have to be used to identify such constant pointer values. To improve detection

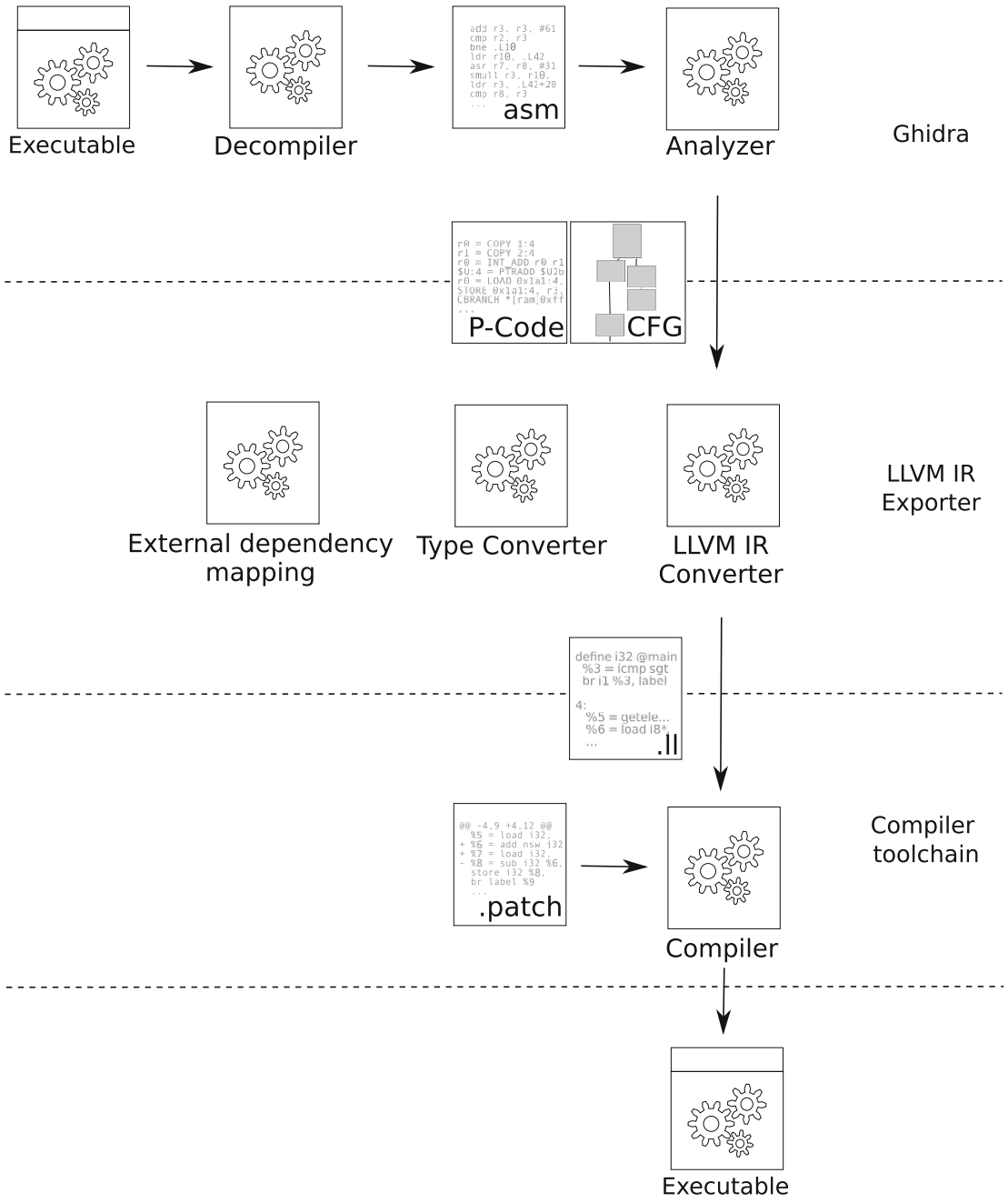

Figure 4.1: Overview of the binary rewriting process

of such values, the image base address can be changed to another higher address such as `0x0ff50000`. While this works great for position-independent binaries, the image base address of a system image used in an embedded device is important because the

compiler may not have generated position-independent code and therefore moving the image base address will cause errors. Such errors can span from simple misidentification of data labels up to a completely different control flow. Additionally, it is important for embedded systems to note that the overall memory space is segmented into different areas and not all of these areas can contain program code. As a result, constant values that are used as pointers and reference an absolute position in the binary, can be easier distinguished from constant values that reference executable code and therefore can be identified correctly in the transformation process. The correct identification of types is very important as it will be discussed in Section 4.1.

4.1 Type Conversion

Ghidra type	Size (in bytes)	LLVM type
`void`	0	`void`
`bool`	1	`i1`
`char`	1	`i4`
`short`	2	`i8`
`int`	4	`i32`
`long`	8	`i64`
`float`	4	`f32`
`double`	8	`f64`
Undefined	$sizeof(Undefined)$	`i<size>`
Array<T>	$n * sizeof(T)$	`[n x T]`
Pointer<T>	4	`T*`
Enumeration	4	`i<size>`
Structure	$sum(struct.elements)$	`{ <type list> }`
Union	$max(sizeof(union.elements))$	`{ <biggest type> }`
FunctionDefinition	4	`(<return> (<params>)*`
TypeDef<T>	0	`T`
String	$strlen(str)$	`[?  x i8]`

Table 4.1: Type Conversion

Converting types from P-Code to LLVM cannot be done trivially in all cases, because Ghidra tries to map all recognized types according to a simplified specification of the C/C++ language, as shown in Table 4.1. Therefore, not only primitive types such as `int` and `char` are present, but also type aliases (`TypeDef`) can exist, which has no representation in LLVM IR. This can partly be a problem, since such type constructs as `TypeDef` cannot be converted and integers have usually a platform depended size. Therefore, when converting types, the length of the individual types has to be considered. A great example for this behavior can be the `Undefined` data types, which can exist in different lengths and represent a type which Ghidra could not recover correctly. Most of the time such instances are limited to the size of a register and can therefore be safely

converted to a native integer type, even if the full length is not needed in that instance. In other cases, the type could be bigger and represent a structure or a union type, which either has to be specified manually or an appropriate process has to be found to identify hand handle the type correctly. As a fallback method such a variable sized type can still be converted into a byte array. But when accessing such a type can be more difficult, as the types of the extracted members have to be casted to a compatible type.

As shown in Table 4.1 the sizes of types in Ghidra are given as bytes, while all types in LLVM IR are using bits to specify their length. Additionally, LLVM only specifies integer and floating-point types as primitive types, other type combinations such as `char` have to be mapped to an appropriate integer type in LLVM. When mapping such types, it is important to note, that while a `char` type in Ghidra has the length of one byte, the LLVM equivalent type would be `i8`, as all integer and floating-point types in LLVM specify their length in bits. Although it is possible to generate type structures in LLVM, the generation of these structures is not as simple as it seems. Dealing with structure and union types can be a bit more difficult, since these types can generally be constructed recursively and therefore a lazy approach must be taken. Such a behavior can be implemented with the LLVM API by first creating a named structure and then resolving all child types, as this ensures that a recursive reference to the same type can be resolved without producing an endless loop. Union types on the other hand, can be a problem, because LLVM does not directly support union types. To be able to handle such union types correctly a structure type should be defined with the length of the biggest containing element of the union [35]. To access the different elements of a union type, additional steps have to be performed as shown in Listing 5. Either a **`bitcast`** with an additional **`load`** expression has to be generated when accessing a component of a union, or alternatively also a **`extractvalue`** with an **`trunc`** could be used if the type can be casted to an array type.

```
; typedef union {
;   char* ptr;
;   long long lValue;
; } example;
;
; example->ptr

%union.example = type { i64 }

%1 = bitcast %union.example* %0 to i8**
%2 = load i8*, i8** %1, align 8
```

Listing 5: Example access of an union type in LLVM

When converting other higher-level types such as arrays and pointers the conversion from the Ghidra type to the LLVM IR type can be considered easier than it is the case with

unions or structs, because these types cannot be specified in a in a recursive way without caching the declaration of the type in LLVM IR. For these types, which are denoted as `TypeName<T>` in Table 4.1, the containing type is `T` and has depending on the type different meanings. For arrays it is the type of the containing elements, for pointers it is the type which is pointed to and for type definitions it is the underlying type, which should be used instead of the type definition in LLVM IR. The type `Enumeration` is an enumeration type that Ghidra has recovered and has to be converted to the corresponding integer type when used in LLVM IR, since there is no support for such types in LLVM IR either. Similar to the C language, function definitions are function pointers that have both, a return type and parameters and therefore are converted in a similar fashion.

But not all types that are listed in Table 4.1 can easily be converted to their respective counterparts in LLVM. One of these types in a `String` data type in Ghidra. This datatype does not specify a length by itself and can only be converted to LLVM IR if the data is also known at the time of the conversion process. When converting the referenced data has to be queried and the length of the string has to be computed dynamically. Although this allows the creation of globally stored, or local string constants, working with these values is not trivial, because in most cases an implicit type cast is needed to convert a character array to a character pointer. Furthermore, it has to be noted that the existence of debug symbols can greatly increase the amount of complex types that can be recovered from the binary by Ghidra. If no such symbols are present most complex types will be represented as pointer operations in P-Code, which requires the generation of more complex typecasts, but may not require a sophisticated type conversion algorithm.

4.2 Basic Blocks

When converting basic blocks from the control flow graph, note that the semantics between basic blocks from the high-level or low-level P-Code structure are different from the semantics of basic blocks in LLVM. In LLVM it is required that a block is terminated by a list of predefined instructions (terminator instructions), which is not always the case in the control flow graph structure provided by Ghidra [36]. Because of the default behavior in Ghidra, such terminator instructions are not needed, as the control flow will be implicitly transferred to the next block in the control flow graph, if only one child block exists. If no such trailing branch is found in the block, an implicit branching instruction (`br label %NEXT_BLOCK`) has to be inserted at the end of the block where `%NEXT_BLOCK` will be the label of the next block that is referenced in the control flow graph. In most cases a high-level P-Code block is terminated by a `BRANCH`, `CBRANCH` or `BRANCHIND` as the control flow branches at that position. High-level constructs such as loops, and branching statements (`if`, `switch`, `goto`) will produce such basic blocks and can therefore be detected by the decompiler when analyzing the control flow. While such blocks cannot only miss an explicit terminator instruction, these blocks can also contain a series of `MULTIEQUAL` operations that have to be handled with care, as explained in Section 4.8.1. The contents of Listing 6 can be seen as example of a series of blocks that do not contain these discussed terminator instructions and a `MULTIEQUAL` operation.

```
        $Ud080:1 =  COPY (const, 0x1, 1)
        CBRANCH *[ram]0xff51026:8 , $Ud080:1

    LAB_0ff51019:
        local_c = COPY (const, 0x2, 4)

    LAB_0ff51026:
        local_c = MULTIEQUAL local_c, r0
```

Listing 6: P-Code blocks without explicit terminator

When transforming basic blocks, the order in which the blocks are processed is important, because depending on the control flow of the function different varnodes of P-Code operations can have dependencies to one, or more predecessor (parent) blocks. Although processing blocks in ascending order seems to be a suitable solution, it can lead to dependency problems later on, because the location of a block, or the order of the blocks in the Ghidras data structure does not necessarily depict the logical control flow of the function. Therefore, to minimize possible dependency problems the blocks are sorted such that the blocks with the least amount of untransformed parent blocks is chosen as next block for the transformation.

Algorithm 4.1: Basic Block ordering

Input: Control flow graph CFG

```
1  allProcessedBlocks ← ∅;
2  edgeList ← ∅;
3  currentBlock ← CFG.entryPoint;
4  while currentBlock is not null do
5      foreach Block parent_i ∈ currentBlock.parents do
6          if not allProcessedBlocks.contains(parent_i) then
7              edgeList.append(parent_i);
8          end
9      end
10     processedBlocks ← convertBlock(currentBlock);
11     allProcessedBlocks.addAll(processedBlocks);
12     edgeList.removeAll(processedBlocks);
13     foreach Block block_i ∈ edgeList do
14         block_i.removeDependency(currentBlock);
15     end
16     edgeList.sortBy(_.openDependencies);
17     currentBlock ← edgeList.next();
18 end
```

While an algorithm like shown in Algorithm 4.1, is able to process blocks as discussed, it is important to also define at which point two blocks are the same, because Ghida cannot only generate multiple P-Code operations per assembly instruction, but these P-Code operations can also manifest themselves in multiple different basic blocks. Such blocks have the same starting and ending address, but differ in their contents, the P-Code operations. Therefore, also the contents have to be minded in the `contains` function of the set operations otherwise these additional blocks can get lost in the process and lead to a corrupted control flow after the function has been transformed to LLVM IR. Besides this issue, it should also be noted that the first block of a function in LLVM IR is not allowed to have any incoming edges, but since Ghidra will generally generate such a control flow graph and therefore it is not necessary to handle this issue any different from handling the generation of other basic blocks. Such empty blocks will only contain the implicitly generated trailing instruction. Since the algorithm only looks at the number of dependencies after they have been processed, it is still possible that a block with at least one untransformed parent will be processed if the control flow graph contains any loops. For such a case the algorithm must be either adapted to also account for such constructs in the control flow graph, or the `convertBlock` function is able to handle such cases by simply transforming the block if an unresolved dependency is encountered. Although, it would be possible to process all the basic blocks of a function in random order with such an approach, it has to be minded that when transforming a block, a certain state has to be created to keep track about special constrains such as the `MULTIEQUAL` constraint or the trailing terminator instruction. Depending on the implementation creating such states can be more compute intensive then working with sorted queues or lists as proposed in the Algorithm 4.1.

After the transformation of the basic blocks in Listing 6 the equivalent LLVM IR blocks should be equivalent to the LLVM IR blocks shown in Listing 7. As discussed, such a transformation has to explicitly insert a trailing terminator instruction in the block at the label `LAB_0ff51019` to produce valid LLVM IR code.

```
%"u_d080:1$0" = i1 1
br i1 %"u_d080:1$0", label %LAB_0ff51019,
    label %LAB_0ff51026

LAB_0ff51019:
%"local_c$1" = i31 2
br label %LAB_0ff51026

LAB_0ff51026:
%"local_c$0" = phi i32 [ 0, %entry ],
    [ %"local_c$1", %LAB_0ff51019 ]
```

Listing 7: LLVM IR basic blocks

4.2.1 switch statements

Switch statements or BRANCHIND operations are more complex to process as a simple if statement, because in contrast to a CBRANCH operation a BRANCHIND operation can have numerous basic blocks. Ghidra collects this information in form of jump tables in an internal data structure, which is referenced by the BRANCHIND operation. This jump table contains an exhaustive list of input values and references to the target basic blocks. Although, LLVM has support for a **switch** instruction, a trivial transformation from the BRANCHIND is not easily possible, because a jump table in Ghidra does not contain a default branch and has to be computed beforehand. Computing such a default case can be considered rather trivial, since a default case can be detected by comparing the number of referenced blocks with the number of labels. Since a default case does not have a label, such an entry can be taken from the jump table and used as default case for the **switch** expression. If no such a default case is present an additional basic block with an unreachable trailing instruction has to be generated, as the **switch** expression requires a default label. In order to generate these switch operations as similar to the original code, the order of the blocks can be considered important and such a generated default label will be placed after the last label that is referenced by the switch statement. While this block order ensures that the relative order remains the same, it also ensures that jump tables present in the assembly code can be recompiled closer to the original than if the blocks had a completely random order. To also aid the readability of the transformed code, all basic blocks that are referenced by a switch statement are renamed to their respective names in Ghidra, which are named after the naming convention shown in Listing 8, where `switch_address` is the address of the BRANCHIND operation and `switch_label` is the label of the case in the jump table.

```
switchD_<switch_address>_caseD_<switch_label>
```

Listing 8: Ghidra switch blocks naming convention

4.3 Static Single Assignment form

Converting P-Code to LLVM IR can be a tricky task, because P-Code is not in a valid static single assignment (SSA) form and therefore operations can overwrite variables. While this behavior can more accurately represent the actual register status of the assembly code, it is not allowed in a valid SSA form and therefore needs to be handled explicitly in a transformation from P-Code to LLVM IR. In Listing 9 we can see an example transformation from P-Code to LLVM IR that tries to resolve this issue by renaming the variables in such a way that the result of the transformation yields a valid SSA form. Such a transformation can be accomplished by keeping track of all variable assignments with a simple counter. Each time a variable gets assigned the counter will be increased and only the occurrence with the highest counter is used as a parameter for

any operation. This ensures that no variable with the same name is overwritten, as well that only the latest assigned value is used for any operation.

```
r0 = COPY 1:4                    %r0.0 = i32 1
r1 = COPY 2:4                    %r1.0 = i32 2
r0 = INT_ADD r0 r1               %r0.1 = add %r0.0 %r1.0
```

Listing 9: Example transformation from P-Code to LLVM IR

Besides the restriction of having only one assignment for each variable the SSA form, the existence of ϕ-expressions can also be seen as problematic, because no low-level P-Code operation exists that is similar to the expression. Although there is no low-level P-Code operation, the high-level P-Code operation `MULTIEQUAL` [37] is very similar to ϕ-expressions. In contrast to a ϕ-expression a `MULTIEQUAL` operation is not as restrictive, as it can occur at any point in a block, while all ϕ-expressions must dominate all other operations in the LLVM IR. This of course means that if a single block in the control flow graph has multiple input edges a reordering of all P-Code operations might be necessary, otherwise such a property cannot be guaranteed. Besides reordering the appropriate operations, ϕ-expressions should be generated in a lazy fashion, as it is possible that not all input edges of a block are already converted and therefore the exact reference of the P-Code variable is not known at the time. Besides these considerations a `MULTIEQUAL` can also have other problems while transforming from P-Code to LLVM IR, which are discussed in Section 4.8.1.

But also other P-Code operations, such as `COPY`, should be handled with care, since constructs such as `%1 = %0` are technically not forbidden by a sound SSA form, but nevertheless cannot be produced by the standard IRBuilder implementation of the LLVM framework. This is mostly the case because such expressions can be considered useless. Such a variable can only be assigned once and such an expression will effectively just produce two variables with the same content, which means that an optimizer can merge these two variables and therefore not only save a valuable register, but also reduce the number of potential instructions that are generated by the code. One possible way to solve this problem of generating such useless constructs makes it necessary to inspect a `COPY` operation, and if such an operation would yield a `%1 = %0`, copy the value "virtually" to the output varnode without generating LLVM IR code at all. The effect of such a procedure is that the next input will reference `%0` instead of `%1`, which is essentially the same, as it cannot be overwritten in a valid SSA form. This of course means that we have to track not only the names of the varnodes as discussed beforehand, but also their last referenced value. But also using the intrinsic function `llvm.ssa.copy` can be seen as valid way of dealing with `COPY` operations, but since this function is not meant for general use, it should be avoided if possible [17].

4.4 Pointer arithmetic

In both P-Code and LLVM IR, the handling of pointer arithmetic differs from each other. While both sides support the manipulation of pointers with integer arithmetic, also dedicated operations to perform different kinds of pointer arithmetic exist on both sides. These operations should be preferred over integer arithmetic, because the pointer operations can be encoded in a platform and type agnostic way. Encoding pointer arithmetic with such instructions also has the advantage that the analysis can be designed simpler as no distinction between normal integer and pointer arithmetic has to be done.

```
$U0002b:4 = COPY 4:4
$U08380:4 = PTRADD $U0002b:4, $Ufffec:4 , 0x4:4
r0 = LOAD 0x1a1:4, $U08380:4
```

Listing 10: P-Code array access

Array access, as shown in Listing 10, utilizes the `PTRADD` operation, which is used to compute the pointer of an array element. In the example, the 4th index of an array that is stored on the stack is loaded into the register `r0`. While this P-Code operations utilizes the attached type information to infer the size of the elements in the array, the P-Code representation encodes this as third parameter. Although the `PTRADD` operation can be considered powerful, it still lacks the ability to access data structures with different sized types [37]. The `PTRSUB` operation can be used to express such a pointer computation and is the more generalized version of the `PTRADD` operation, because it only needs a base address and a byte offset to compute the pointer. Although in most cases varnodes are used to identify the base pointer of such an operation, it is also possible that only constant values are used for the calculation. Which means that constructs, such as `PTRSUB 0x0 0x010444` and `PTRSUB sp 0xffffff4`, are valid constructs and have to be transformed correctly to avoid breaking the program. While the first one refers to an absolute position in the volatile memory, the second P-Code operation is used to compute the storage location on the function stack. When transforming these P-Code operations to LLVM IR, the matching varnodes have to be computed in order to generate valid LLVM IR code. This is necessary, because while it is possible in Ghidra to refer to a memory location by a constant value, in LLVM no such overall memory space exists and all operations on any memory segment can only be done over valid LLVM value references. It is important to note that the result of these operations are pointer types, which can also be manipulated with integer arithmetic afterwards by P-Code. For more complex pointer operations the offset parameter of the `PTRSUB` operation is calculated with integer arithmetic as exact information of types is not preserved by the compilation and has to be filled in by Ghidra.

In LLVM IR the **`getelementptr`** instruction can be used to perform all kinds of pointer arithmetic, which is not limited to array access as it is the case with the `PTRADD` operation in P-Code. Therefore, the compiler also has to know the exact types that are used by this expression to perform the correct arithmetic operations. Similar to

```
%11 = i32 4
%12 = getelementptr inbounds [5 x i32],
    [5 x i32]* %2, i64 0, i64 %11
%13 = load i32, i32* %12, align 4
```

Listing 11: LLVM array access

P-Code this expression yields a pointer type, which has to be loaded explicitly afterwards. But compared to the previously mentioned P-Code operations, the **getelementptr** expression uses only indices to access the elements of an array or structure. Therefore, a simple transformation between a PTRSUB operation is not easily possible, because an attached type information is needed or the types have to be casted beforehand to a byte array like structure that supports byte precise indexing. In Listing 11 the transformed P-Code utilizes a **getelementptr** expression to compute the pointer that will then afterwards be loaded via a **load** expression to access the value [38].

4.5 Data Labels

Data labels in Ghidra can be seen as special labels that indicate at which addresses are referenced by P-Code operations in the binary. These labels cannot only be renamed, like local variables, but can also contain a higher-level type, which can indicate which kind of data is stored at the location. Data labels usually do not have a specified size if no concrete data type is given and are not limited to occur in data only sections such as `.bss` or `.rodata` sections of a typical ELF binary. It is also possible that the `.text` section has such labels in between code segments, or that that data labels are created at specific addresses, that are not part of any section in the ELF binary. For embedded system images, this is a strong indication that such an address is a hardware-specific address, which can be used to communicate with other hardware. Not only can the serial console be such a hardware, but also other peripheral devices for different tasks such as networking, Bluetooth, sensors can be located at such addresses. To avoid potential problems, such addresses should be marked as `volatile` in Ghidra to ensure that when the corresponding **load** / **store** expressions are generated in LLVM IR are also marked as volatile. Because this also ensures that neither the optimizer, or the compiler treats these operations as useless and removes the access from the final binary.

When working with P-Code, such data labels identify global stored variables or data structures regardless of the section they are stored in. In case of the `.rodata` section, or any other section that is marked as read-only section, such data structures can have a constant value. This means that the values cannot be changed in any way and therefore all bytes can be copied and marked as constant in the resulting LLVM IR code. Due to these limitations, not all data labels can be converted in the same way, as the actual section permissions must be taken into account. While marking some global symbols as constant values can already be considered as an optimization, it cannot only help to reduce the overall binary size, because similar symbols can be merged, but also helps

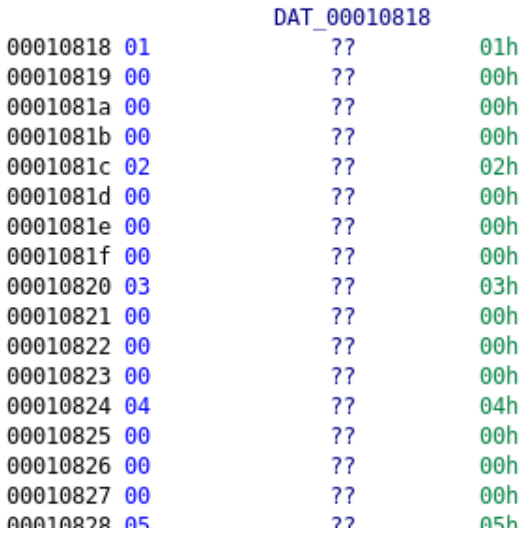

Figure 4.2: Unidentified data label in Ghidra

the compiler in generating better code overall. In order to generate the correct type of global variables or constants the right LLVM IR identify have to be chosen. Most of the time such global variables will be privately scoped and contain the `unnamed_addr` identifier, as we do not have to restrict the address of global symbols in most cases when recompiling a binary. Any other linkage then private is only needed if the global symbol is external. But when converting such data labels into global symbols, it is not only important to choose the right properties for such symbols in the LLVM IR code, but also to make sure that the symbols are sized correctly or even have the correct type. If such data labels are undefined, as shown in Figure 4.2, it can be the case that data can be missing from the transformed binary, if the data label is contained in the `.rodata` section. But also, data access outside the memory region of the global symbols is possible, which can lead to unexpected behavior and cannot only end in segmentation faults, but can also lead to data corruption. To solve this issue, all data labels that are not identified at all by Ghidra can be sized in such a way that they span across the free memory space until the next data label. As type a normal byte array with the determined size has to be chosen and then type casted as needed in the rest of the program where this symbol is used. However, other analyses or algorithms can also be used, because this described algorithm can be considered as best effort approach. Such analyzers could for example make use of further static analysis to identify functions such as `memcpy`, `strlen` or even `printf` to infer the type or the size of a local or global symbol. As shown in Listing 12 the function call to `memcpy` can be used in such an analysis to limit the size of the data label `DAT_00010818` to 32 bytes.

While such an approach can be chosen for dynamically linked binaries, this is not easily possible in static binaries, because these functions are contained in the binary and have to be identified beforehand if no debug symbols are present. In dynamically linked binaries

```
int array [8];
memcpy(array, &DAT_00010818, 0x20);

for (int i=0; i<8; i=i+1)
    printf("%d\n", array[i]);
```

Listing 12: C-Style pseudo code accessing a global symbol

only the imported symbols have to be considered. Additionally, this issue can still occur in binaries that are compiled with full debug information, because it is possible that some data labels have been generated or were modified by the optimization steps of the compiler or directly assembly was used.

4.6 Stack

Correctly handling stack operations when converting P code to LLVM IR can be difficult because the concepts that the two internal representations use to represent operations are very different. In P-Code a stack is just a different memory region and therefore a different memory space id will be used for addresses and varnodes that reference some space in the stack memory region. In low-level P-Code stack values are getting pushed into the stack region via an explicit `STORE ram(stack_ptr)`, where `stack_ptr`, is the calculated pointer to the stack. In high-level P-Code, also plain `COPY`, `PTRSUB` or `PTRADD` operations are used to influence the stack. While a `COPY` is used to copy data to/from the stack, the other operations may indicate that a complex data structure such as structs or arrays is being accessed. To either copy a concrete value to/from the stack or reference the stack via a pointer that can then be used later, a number of operations in LLVM IR are required. It is also possible for `INDIRECT` operations to refer to one or more values on the stack when a function or other assembly instruction may have affected the values on the stack in some way as a side effect. In all these cases, the matching high-level variable must be correctly identified, since it is referenced by an address. Otherwise, it is not possible to create an accurate representation of the matched value in LLVM IR using multiple trivially generated **`alloca`** expressions for these variables on the stack. Without the appropriate debug symbols, functions that perform more or less complex operations on values stored on the stack often use more variables than in the original source code, as shown later in Listing 13. Therefore, any optimization that changes the order or the way variables are stored and that the compiler applies to the stack area can lead to erroneous behavior, since the compiler is not only able to reorganize the stack to save space, but also to move **`alloca`** instructions into registers.

Ghidra does provide a stack frame data structure that lists local and stack variables, but it should be mentioned that the mapping in this structure seems to be incomplete as only symbols of identified variables are mapped in this structure and sometimes it is necessary

```
struct String {
    bool initialized;
    unsigned int size;
    char content[64];
};

void function(char *in) {
    struct String str;
    str.initialized = true;
    str.size = strlen(in);
    memcpy(in, str.content, str.size);

    some_other_function(&str);
}
```

Listing 13: C-Style pseudo code utilizing the stack

to manually correct the identified stack frame. If the binary contains debug symbols or all variables stored on the stack have been correctly identified, a simple **alloca** expression can be generated for each symbol, since no access across multiple symbols should or is intended when running the program with expected input parameters. But since not all functions can be handled by such an approach, it is easier to just view the whole stack region as a single byte array and extract all variables from this array if they are accessed. While this solves many problems, such as the need to correctly identify high-level types, it also prevents the compiler from optimizing many stack operations. Although the relative order of variables on the stack region cannot change, some pointer arithmetic is needed to build the correct access pointer to the stack region. Primitive types, arrays and also structures can then be stored in the stack region via the same means as shown in Listing 14. Another advantage of this approach is that all stack pointer operations can be transformed without requiring any correction, since the stack layout is exactly the same as in the original function. However, this can also be seen as a disadvantage.

As shown in Listing 14 the decompiled and transformed code does not have a single struct data type, as it is the case in the original source code. Instead of a single data type Ghidra managed to detect multiple local variables (`local_c`, `local_50`, `local_54` and `auStack76`) that are used to store data on the stack. While this is not wrong, it also shows that it is easily possible to produce multiple different stack allocations for data that should logically be in a single stack allocation. When using the discussed approach to handle stack variables, a set of expressions must be created to access the variable on the stack. First a **getelementptr inbounds** expression is generated to calculate the pointer offset, which is afterwards casted to the correct type with a **bitcast** expression. Finally, accessing a variable in this approach requires a **load** expression, as in other approaches that use only **alloca** expressions. The resulting variable can then be used like any normal variable, since it contains the actual value instead of a pointer to the

```
; Function Attrs: nounwind
define void @function(i8* %param_1) local_unnamed_addr #0 {
entry:
    %"__Stack[]__" = alloca [76 x i8], align 1
    %"local_54$0" = getelementptr inbounds [76 x i8], [76 x i8]* %"__Stack[]__", i32 0, i32 0
    store i8 1, i8* %"local_54$0", align 1
    %"local_c$0" = getelementptr inbounds [76 x i8], [76 x i8]* %"__Stack[]__", i32 0, i32 72
    %"local_c$1" = bitcast i8* %"local_c$0" to i8**
    store i8* %param_1, i8** %"local_c$1", align 8
    %"local_50$0" = call i32 @strlen(i8* %param_1)
    %"local_50$1" = getelementptr inbounds [76 x i8], [76 x i8]* %"__Stack[]__", i32 0, i32 4
    %"local_50$2" = bitcast i8* %"local_50$1" to i32*
    store i32 %"local_50$0", i32* %"local_50$2", align 4
    %"auStack76$0" = getelementptr inbounds [76 x i8], [76 x i8]* %"__Stack[]__", i32 0, i32 8
    %"auStack76$1" = bitcast i8* %"auStack76$0" to [64 x i8]*
    %0 = bitcast [64 x i8]* %"auStack76$1" to i8*
    %"local_c$2" = getelementptr inbounds [76 x i8], [76 x i8]* %"__Stack[]__", i32 0, i32 72
    %"local_c$3" = bitcast i8* %"local_c$2" to i8**
    %"local_c$4" = load i8*, i8** %"local_c$3", align 4
    %"local_50$3" = getelementptr inbounds [76 x i8], [76 x i8]* %"__Stack[]__", i32 0, i32 4
    %"local_50$4" = bitcast i8* %"local_50$3" to i32*
    %"local_50$5" = load i32, i32* %"local_50$4", align 4
    %"unused$0" = call i8* @memcpy(i8* %0, i8* %"local_c$4", i32 %"local_50$5")
    %"local_54$2" = getelementptr inbounds [76 x i8], [76 x i8]* %"__Stack[]__", i32 0, i32 0
    %"local_54$3" = bitcast i8* %"local_54$2" to i8*
    call void @some_other_function(i8* %"local_54$3")
    ret void
}
```

Listing 14: LLVM IR code when using a stack byte array

stack.

4.6.1 Variadic functions

Although the chosen approach greatly simplifies the handling of the stack content without compromising the correctness of the program, the handling of variable functions can be problematic. Since variadic functions rely on types and functions built in by the compiler, it is not straightforward to include such types in the manually managed stack space. The `va_list` data type is not only platform-specific, but must also be initialized by the corresponding variadic function (`@llvm.va_start`) and released with `@llvm.va_end`. Because of this behavior, it is not possible to integrate the values into the stack variable without copying the data, which can lead to performance overhead. As shown in Listing 15, an extra stack allocation has to be made, which is labeled `__va_list__` and is just used to capture the list of variadic arguments. While it would be sufficient to extract the arguments form the data structure with **`va_arg`** and assign them to a variable, it may not be sufficient enough if the target varnode storage location is inside the stack area. Therefore, all variables extracted from the variadic arguments are stored again in their respective area on the managed stack area. This leads to a higher usage of stack space, because at least the storage requirements for the `va_list` data structure has to be doubled. Storing elements in the stack array works similar to loading a value from it,

with the only difference that a **store** expression is used instead of a **load** expression.

```
%"__Stack[]__" = alloca [20 x i8], align 1
%0 = alloca i8*, align 8
%__va_list__ = bitcast i8** %0 to i8*

call void @llvm.va_start(i8* %__va_list__)

%"p$2" = va_arg i8* %__va_list__, i32*
%"p$3" = getelementptr inbounds [20 x i8],
    [20 x i8]* %"__Stack[]__", i32 0, i32 16
%"p$4" = bitcast i8* %"p$3" to i32**

store i32* %"p$2", i32** %"p$4", align 8
```

Listing 15: Transformed code handling variadic functions

4.7 Handling Varnodes

Since the semantics of accessing the data stored in a varnode are inherently different from P-code and the variables in LLVM IR, a valid transformation between these two internal representations must account for these differences. While P-code operations do not distinguish between different memory areas, this difference is crucial for LLVM IR, since all load and store operations are explicit. This is not only the case in high-level P-Code and such expressions have to be inserted into the transformed LLVM IR code implicitly.

When retrieving data from a varnode, an algorithm must not only consider block-specific constraints, but also check the address of a varnode to decide how to load the data in an LLVM environment. Therefore, the algorithm has to handle the following storage scenarios separately, as they can all generate different expression when converted to LLVM IR:

- **Global storage**: Since a varnode in global storage is handled very differently in P-Code and LLVM IR an explicit **load** expression is needed, because no respective LOAD operation is present in high-level P-Code, as these global variables can be accessed like any other variable that is in scope. Therefore, the algorithm has not only to check if a reference to a data label already exists, but also has to make sure that the appropriate pointer offset is taken into account. While in Ghidra these data labels are part of a memory section, this is not the case in LLVM IR. This is because a data label is treated as a global variable in LLVM and additional information such as the section name can be specified. This additional information can then later be used by the linker to group these variables to recover this memory section. But before a global varnode can be accessed some needed pointer arithmetic or

load expressions are needed. When generating these instructions, the algorithm must ensure that these instructions do not dominate a `MULTIEQUAL` operation.

- **Stack storage**: As discussed in Section 4.6, the stack is treated as a byte array in this approach,and is therefore subject to similar restrictions on access as varnodes that use global memory. When accessing such varnodes it is important to ensure that the type information is reconstructed, otherwise the representing type will differ from Ghidra and the LLVM IR code.
- **Storage identified by definition**: If a varnode is accompanied by information about its definition, this information can be used to query the exact P-code operation that generated the value. Because identifying the variables this way in addition to checking their addresses it is possible to identify potential problems while the binary is beeing transformed. For example, it is possible to detect if the defining P-Code operation is in another block and then force the other block to be transformed before finishing the transformation of the current P-Code operation.
- **Storage identified by varnode**: But it is also possible for varnodes to have no information attached about their defining P-Code operation. Such varnodes can either be function parameters, constant values or completely unresolved registers. If such a varnode is encountered, the algorithm has to decide if it is legal to insert additional expressions or inline assembly to read from the register or if the transformation should be stopped, as the analysis that was performed beforehand is incomplete or not correct enough.
- **Special operations**: Finally, there are also varnodes that have a special address. For example, accessing the stack pointer register can be such a case. While stack pointer operations can be handled as any other pointer operation, such statements still have to be handled with care to ensure that an access to the stack pointer is generated after all `MULTIEQUAL` operations have been processed.

Similar to retrieving values from varnodes, storing values in varnodes must also respect these constraints and may result in either more or no LLVM IR expressions being generated immediately. Because no other expression is allowed to dominate a ϕ-expression, accessing a varnode in a `MULTIEQUAL` operation must modify the end of the parent block if additional LLVM IR expressions have to be generated.

Besides correctly handling the storage type of a varnode, also the data types have to be considered, because Ghidra will not generate an explicit `CAST` operation for types that are compatible. For example, if a string constant is passed to a function as argument, the string constant will have the type a fixed size `String` datatype, while the function argument has a string data type (`char *`). As far as Ghidra is concerned these two data types are compatible with each other and therefore no explicit type cast is required. But in LLVM IR a fixed sized char array and a char pointer are not compatible with each other and require an explicit pointer cast. To simplify the handling of implicit type

additions, they can be generated directly after the generation of the corresponding access expressions if the type is known beforehand. Otherwise, no implicit type cast is possible, which is especially the case for low-level P-Code operations and can lead to errors when the transformed LLVM IR code is compiled.

4.8 Special operations

All P-Code operations that require special handling in the chosen approach can be considered as special operations, since not only the logic of the operation must be correctly transformed during the transformation, but also the additional tasks must be handled. While the processing logic for varnodes is separate from the logic for P-Code operations, as described in section 4.7, the transformation process for most P-Code operations turns out to be more straightforward. However, this is not always the case for the special operations. Some P-code operations such as. `COPY` cannot be transformed in the same way because, as mentioned above, the concept of copying data from P-code to LLVM IR is different. But also, a `INDIRECT` can be considered as such a special P-Code operation, because most of the time not a single LLVM expression will be generated from the `INDIRECT` operation. A `INDIRECT` operation is used as a signal that a certain operation, like a function call, might have modified some data of a varnode, or that an operation influences multiple varnodes at the same time. In such cases it is entirely possible that not a single P-Code operation depends on these varnodes and therefore can be seen as a rather meaningless feature for the transformation process, but in other cases P-Code operations depend on these varnodes and therefore at least a virtual copy has to be done. Although a virtual copy can trivially solve the problem in most cases, it should be noted that in some cases it is not necessary to generate any LLVM IR expressions for such a copy operation. This is because if the source and destination memory addresses are the same, as would be the case with stack or global variables, creating a pair of load and store expressions from the same address would not make sense. However, while these two P-code operations do not add too much additional logical complexity to the transformation process, the P-code operations in the following sections can be considered more complex to process correctly.

4.8.1 MULTIEQUAL

As already mention in Chapter 2, a `MULTIEQUAL` operation is similar to a **`phi`** expression in LLVM and therefore follows roughly the same rules. Similar to ϕ-expressions the `MULTIEQUAL` operations will be generated at the start of a block and therefore dominate all other instructions in such a block. But while in LLVM the domination property is strictly enforced, it is not necessarily the case as the documentation [37] for the `MULTIEQUAL` operation does not mention such a restriction. Additionally, when accessing varnodes that are stored in the stack or global memory regions these varnodes have to be explicitly loaded in LLVM IR, which can generate **`load`** operations between **`phi`** operations, if not minded. Therefore, in the chosen approach all **`phi`** expressions are

generated in a lazy fashion to delay the generation of the input array as much as possible while also generating all LLVM IR expressions in the same order as they occur in the P-Code representation. While this means that variables will have a known type and all following expressions can be generated, a block with such **phi** nodes are not in a valid LLVM IR form until all **phi** expressions are completed. The completion of the expressions can be done at the earliest when all input blocks have been processed or at the latest when all blocks have been processed. Otherwise, looping constructs can be problematic for the transformation process, because a block may need references to itself. Such an example for partially generate **phi** expressions can be seen in Listing 16.

```
LAB_00001888:
    %"DAT_200003b4$0" = phi i32
    %"DAT_200003b0$1" = phi i32
    %"UINT_20000398$0" = phi i32
    store i32 %"DAT_200003b4$0", i32* @bss_200003b4, align 4
    ...
```

Listing 16: Partial generated ϕ-node

Besides handling a lazily generated ϕ-node, also the domination property has to be minded, because as discussed in Section 4.7 some varnodes may need additional expressions in LLVM IR. For example, if a **phi** expression has a varnode as output that represents a global symbol, as shown in Listing 16, an additional **store** expression is needed. The creation is then delayed until all MULTIEQUAL operations have been processed. But if any varnodes are used that require additional expressions, these have to be inserted in the matching parent block, because the output value of a ϕ-node will assume the value of the variable from the incoming edge.

But not all MULTIEQUAL operations that are generated by Ghidra during the analysis phase will be transformed to an equivalent **phi** expression. For example, Ghidra will generate MULTIEQUAL operations for accessing variables on the stack or global variables. If the resulting varnode and all input varnodes of such an operation target the same storage address, then the transformation process can simply discard the operation, because it would just generate multiple load expressions at the end of the previous blocks and a store expression after all other MULTIEQUAL operations. Because of how global and stack related varnodes are handled by the prototype discarding does not produce an error as the access to such a varnode already generates a **load** operation in the transformation.

4.8.2 CALLOTHER

When processing CALLOTHER operations the implementation of a operation can be very different depending on the first parameter of the operation, which indicates its type. As shown in Table 4.2 some of these functions can be implemented with normal LLVM IR code, while others have will generate inline assembly, because these functions represent platform specific code that is generated by some higher-level constructs. While values such as

`output` und `input<n>` represent the output and input varnodes of the operation, inline assembly instructions use `$0` as placeholder for the register, which will then be assigned by the compiler. All operations listed in Table 4.2 except `hasExclusiveAccess` will be processed when encountered. `hasExclusiveAccess` on the other hand will be ignored, because this specific functionality indicates access to a varnode which should happen atomically. Because atomic assembly instructions are not well translated to P-Code, these cases will be handled separately as discussed in Section 4.8.3.

Name	Code	Generated code
software_interrupt	0x10	`svc #input1`
enableIRQinterrupts	0x1E	`cpsie i`
disableIRQinterrupts	0x21	`cpsid i`
isIRQinterruptsEnabled	0x24	`mrs $0, primask`
hasExclusiveAccess	0x26	—
isCurrentModePrivileged	0x27	`%output = COPY %input1`
DataSynchronizationBarrier	0x2E	`call @llvm.arm.dsb(%input1)`
WaitForEvent	0x30	`wfe`
WaitForInterrupt	0x31	`wfi`
Ins.Sync.Barrier[1]	0x33	`call @llvm.arm.isb(%input1)`
DataMemoryBarrier[2]	0x5D	`mcr p15,0x0,$0,cr7,cr10,0x5`
getMainStackPointer	0x110	`mrs $0, msp`
getCurrentExceptionNumber	0x113	`mrs $0, ipsr`
getProcessStackPointer	0x111	`mrs $0, psp`
setProcessStackPointer	0x117	`msr psp, $0`
read_volatile	0x11a	**`load volatile`**
write_volatile	0x11b	**`store volatile`**

Table 4.2: Excerpt of generated code by `CALLOTHER`

Besides the `CALLOTHER` operations listed in Table 4.2, also other ones exist that cannot simply be transformed into LLVM IR. In some cases, like atomic load/store operations, the P-Code represents the logical part of the contained assembly instructions and important features are abstracted away. This can lead to an incorrect transformation. However, even when accessing hardware-specific functions, such as accessing the `CONTROL` register on an ARM Cortex CPU processor, the generated P-code represents the logical flow of information, as shown in Listing 17. In this Listing the current contents of the `CONTROL` register are retrieved and the stack mode of the processor is switched to the `PSP` (process stack pointer) register. Therefore, not only the control flow has to be changed accordingly, but also the logical changes to the individual bits of the register have to be parsed and translated back into their respective assembly instructions. This is especially important for code segments that are shown in Listing 17, because in assembly code such operations

[1] InstructionSynchronizationBarrier
[2] coproc_moveto_Data_Memory_Barrier

would have been performed by 4 assembly instructions and introducing branching code may hurt the performance in such cases.

```
char cVar1 = isThreadModePrivileged();
char cVar2 = isUsingMainStack();
bool bVar2 = (bool)isCurrentModePrivileged();

if (bVar2) {
    setThreadModePrivileged(cVar1 == '\x01');
    bVar2 = (bool)isThreadMode();
    if (bVar2) {
        setStackMode(1);
    }
}
```

Listing 17: C-Pseudocode for accessing the CONTROL register

4.8.3 Atomic operations

Atomic operations can be troublesome when converting P-Code to LLVM IR, because both low and high-level P-Code do not have a notion of atomicity and therefore the assembly code has to be inspected to be able to identify such operations. Instructions that load values atomically from main memory, such as `ldrex`, are often translated into a direct `LOAD` operation in P-Code and can therefore easily lead to incorrect results in a multi-core environment. But other instructions, like `strex`, can be translated in a way that the resulting P-Code pictures the logical changes of the code, as shown in Listing 18. Although the resulting P-Code is not wrong in any sense, it cannot be transformed to LLVM IR without any post processing at all, because multiple P-Code operations have to be translated into a single assembly instruction and the resulting control flow graph has to be adapted, such that the branching and store operations are not present in the final output. Therefore, the functions should be scanned for the occurrence of such instructions and the resulting P-Code operations should then simply be discarded and replaced by a custom function that produces the appropriate LLVM IR expressions.

Although such a solution can be seen as acceptable, it is far from optimal, as it is most likely the case that not only inline assembly, but also platform specific LLVM intrinsic functions have to be used in that process. Not only can this prevent the optimizer from being as effective as possible, but it also means that when patching sections of code that access such variables, synchronization and atomic primitives must also be used to ensure data consistency. Another way to solve this problem would be to use the commands present in the binary to determine what atomic order the compiler might have used, and then convert the corresponding expressions **`load atomic`** and **`store atomic`**. But this approach can be more error prone, because the developers can also explicitly use synchronization primitives and therefore produce a series of instructions that cannot be mapped to an appropriate ordering pattern. Nevertheless, in such cases it is still possible

```
LAB_0xff5019c.0:
    $U6a800:1 = CALLOTHER "hasExclusiveAccess", r2
    r0 = COPY 1:4
    $U6a800:1 = BOOL_NEGATE $U6a800:1
    CBRANCH *[ram]0xff501a0:4, $U6a800:1

LAB_0ff5019c.1:
    r0 = COPY 0:4
    STORE 0x1a1:4, r1, r0

LAB_0ff501a0:
```

Listing 18: P-Code for `strex r0, r1, [r2]`

to fall back to the simpler approach, which is to generate these synchronization and atomic load/store primitives exactly the way they are in the original application [39] [40].

4.8.4 ARM EABI functions

Various ARM EABI functions consist of highly optimized assembly code to create assembly that is as efficient as possible and therefore a transformation may not be possible. Such functions are used by different compilers to implement various arithmetic functions that are not natively supported by the instruction set of the targeted ARM processor. An example for such operations would be floating-point operations, because an ARM processor is not required to support floating-point operations, but also handling lager integer divisions or multiplications can utilize these functions. To simplify the generated LLVM IR code these functions can be reduced to their respective LLVM instructions and ignored by the transformation process. But this does not only require the correct identification of these function in Ghidra, but also requires special handling in the transformation process. Especially when encountering functions, such as `__aeabi_idivmod`, which has a more complex return type then a simple primitive value. In this case not only one, but a series of LLVM IR instructions have to be generated to model the effect of the function and pack the resulting values together into the correct type, as shown in Listing 19. First the two LLVM IR instructions have to be generated that represent the functionality of the ARM EABI function, which are an integer division that yield not only the quotient, but also the remainder. The reminder is then calculated with the **`srem`** expression. After that the both results are extended to the right size, such that they can be merged into one type that is equivalent to the result value of the EABI function. While compilers may generate access to the function when only performing a modulo operation, it is important to generate both operations and merge the results correctly, because it is possible that a wrong transformation can lead to a corrupted state in the final program.

```
; div + mod
%quotient  = sdiv i32 %"u_3580:4$0", %"__n$0"
%remainder = srem i32 %"u_3580:4$0", %"__n$0"

; extend to i64
%"quotient$1" = zext i32 %quotient to i64
%"remainder$1" = zext i32 %remainder to i64

; pack result together
%"remainder$2" = shl i64 %"remainder$1", 32
%result = or i64 %"quotient$1", %"remainder$2"
```

Listing 19: Transforming `__aeabi_idivmod` to LLVM IR

4.9 Dynamically linked ELF binaries

When handling dynamically linked ELF binaries it may be necessary to infer the correct version of the standard library that is used by the binary, as it could otherwise result in a lot of errors at the linking stage. Besides glibc [3], also musl [4] or newlib [5] are examples for alternatives that can be used for resource constraint systems. While glibc can be considered to have good binary and source level forward and backward comparability, this might not necessarily be the case for other implementations. Such implementations are not necessarily compatible with glibc on a binary level or source level and therefore when trying to link the application errors can occur. Generally, there are different approaches for re-linking binaries to their original libraries, including the standard library, but in this thesis, we are only focusing on two main differences that can be applied in different scenarios:

Linking with original sections: When linking a binary with an unmodified entry point and other relevant sections such as `.init`, `.fini`, `.init_array` and `.fini_array` it is important to keep in mind that linking in such an approach is not trivially possible. Not only will a linker insert an appropriate entry point for the binary (`_start`), but also other functions that are responsible to setup the environment and execute global initializers. These functions are contained or referenced in the previously mentioned sections. For older binaries it is also possible that such global constructors and destructors are residing in the `.ctor` / `.dtor` sections, which has to be minded [41]. When building and linking an application with gcc and glibc, such global constructors and destructors can simply be created by specifying the attribute `__attribute__((constructor))` or `__attribute__((destructor))` in the function declaration. In order to correctly link a binary that contains such functions, the appropriate sections have to be reconstructed, and the linking process has to be customized extensively, because otherwise it

[3] https://www.gnu.org/software/libc/
[4] https://www.musl-libc.org/
[5] https://www.sourceware.org/newlib/

is to be expected that the symbols of these functions or sections either conflict with each other, or that the resulting ELF executable will not work as before.

Rewriting the entry point: As an alternative to the customization of the linking stage, as mentioned in the previous paragraph, the binary can also be re-build with a slightly modified version of the entry point to allow for easier linking, but therefore also needs additional analysis for it to work properly. Since the entry point will be generated by the chosen standard library, such as glibc, it can be omitted from the final export. But it has to be ensured that the entry point is indeed generated by a standard library and not hand crafted, to avoid further complications at runtime. In such an analysis step also the used parameters of `__libc_start_main` can be inferred, if not already done by Ghidra. While it does not seem very important, it is still required for future analysis, because functions like `_libc_csu_init` and `_libc_csu_fini` are already contained in the binary, but not really needed anymore and can therefore be discarded. Besides these symbols also the initializer sections (`.init`/`.init_array`) should be handled with care, as they must be handled differently in LLVM. In LLVM the global variable `llvm.global_ctors` is responsible for mapping these sections and therefore any global constructor function that is contained in the initializer section must be appended to this global variable. At this point, it should be noted that the order and therefore also the priority should be the same as in the initializer sections, otherwise the correctness cannot be ensured. Additionally, the same has to be done with the destructor sections (`.fini`/`.fini_array`). But in this case, it should be noted, that the functions appended to the global variable `llvm.global_dtors`, will be processed in descending order. After these steps the resulting application should have correctly mapped all the global constructors / destructors and a named `main` function, which means that it should be able to link the code with any standard library as long as there exists source code comparability.

4.9.1 Weak symbols

When working with binaries that have been compiled by GCC it is possible to encounter weak external symbols even when the application does not explicitly use them in any way. Such symbols can be present to support older versions of glibc or to provide additional features if certain dependencies are present. One such external weak dependency may be `__gmon_start__`, which should only be needed if the application is compiled with support for the GNU profiling tool (gprof), but is still referenced in the symbol table of an ELF binary [42]. Additionally, function references such as `__cxa_finalize@GLIBC_2.4`, `_ITM_registerTMCloneTable` and `_ITM_deregisterTMCloneTable` can be found in such binaries. Such functions should be handled more carefully when exporting a binary, because they not only need the correct linkage type set to `extern_weak`, but also have to be identified correctly beforehand. Currently the ELF parser of Ghidra does not support the identification of weak symbols and therefore the ELF headers must be parsed manually or other tools

have to be used to identify such symbols correctly [43]. If these weak symbols are not defined correctly linking will likely fail with either conflicting or missing symbol errors, because such weak symbols can be referenced in an if statement to check if the weak symbols is implemented or not [17].

4.10 Embedded system images

Compared to static or dynamically linked binaries, a lot more details have to be considered when processing embedded system images. Ghidra supports loading ELF executables out-of-the-box, which may not necessarily the case with system images, because various different kinds of formats may need different kinds of loaders that have to be provided for the prototype to work. But besides the basic analysis and identifying notable sections in embedded system images, further analysis can be more difficult to accomplish, because ELF executables usually contain some meta information about these sections, which may not be the case for embedded system images. Although it is sometimes possible to export such images as ELF binaries, additional care has to be taken when processing sections such as the interrupt vector table, interrupt handlers context switching functions of the system kernel. These elements can cause multiple problems in the transformation process if not handled separately in the transformation process, which will be discussed in the following subsections.

4.10.1 Interrupt vector table

```
00000000 e0 07 00 20     addr      DAT_200007e0              [0]

00000004 b5 18 00 00     addr      entry+1                   [1]
00000008 f7 3a 00 00     addr      FUN_00003af6+1            [2]
0000000c e1 18 00 00     addr      FUN_000018e0+1            [3]
00000010 00 00 00 00     addr      00000000                  [4]
00000014 00 00 00 00     addr      00000000                  [5]
00000018 00 00 00 00     addr      00000000                  [6]
0000001c 00 00 00 00     addr      00000000                  [7]
00000020 00 00 00 00     addr      00000000                  [8]
00000024 00 00 00 00     addr      00000000                  [9]
00000028 00 00 00 00     addr      00000000                  [10]
0000002c 5d 17 00 00     addr      FUN_0000175d              [11]
00000030 00 00 00 00     addr      00000000                  [12]
00000034 00 00 00 00     addr      00000000                  [13]
00000038 fd 16 00 00     addr      FUN_000016fc+1            [14]
0000003c 00 00 00 00     addr      00000000                  [15]
00000040 75 18 00 00     addr      FUN_00001874+1            [16]
00000044 75 18 00 00     addr      FUN_00001874+1            [17]
00000048 75 18 00 00     addr      FUN_00001874+1            [18]
```

Figure 4.3: Interrupt vector table in Ghidra

The Figure 4.3 shows an excerpt of the interrupt vector table, which starts at the beginning of the hello world Zephyr image. Although this can be considered a typical place for the interrupt vector table on embedded ARM systems, it is also possible to

Type	Priority	Description
IRQ	4	General purpose interrupts
FIQ	3	Fast interrupts
SWI	6	Privileged function call
ABORT	2	Memory or Instruction fetch failure
UNDEF	6	Undefined instruction

Table 4.3: ARM interrupt handler types[45]

change the location of the interrupt vector table by utilizing the `VTOR` register [7]. This is typically done by a bootloader or any other kind of software that is capable in initializing the environment for the main operation system. Depending on the actual hardware that is used, each entry of the interrupt vector table can have a different meaning and therefore also different properties, which have to be considered when generating the functions for handling interrupts. In LLVM IR an interrupt handler can be marked as such by adding the `interrupt` attribute with a value indicating kind of interrupt handler. For this type multiple types are supported by the LLVM toolchain, which are IRQ, FIQ, SWI, ABORT and UNDEF[44].

But even if a specific interrupt handler convention is known to the framework, it doesn't mean that the current code, which is used to process the interrupt, also implements the convention. For example, if the interrupt handler was written in assembly, the compiler will not append an additional prolog or epilog to the function and the author of the code has to make sure that the register values are saved and restored correctly. If not, data can be leaked and the userspace state could be corrupted. Therefore, when transforming the code such properties have to be taken into account. Either the function can be transformed like any other function in the embedded image with the addition of the interrupt attribute or has to be handled like a naked function as described in the Subsection 4.10.2. For deciding which of the two strategies should be used to transform the interrupt handlers, the prototype checks the functions for missing high-level references to registers, which may indicate that registers are used in a way that are not covered by the calling convention. To keep this step rather simple, floating point registers and other special purpose registers will not be minded. While this approach is not perfect, as constructs that do not produce high-level P-Code at all will be missed, it is still sufficient to produce LLVM IR code that reflects the decompiled code in Ghidra.

4.10.2 Handling naked functions

While transforming complex binaries or embedded system images it is possible that functions are encountered which cannot entirely expressed in high-level P-Code without missing references, operations or not capturing all operations accurately. For example, the function `z_arm_pendsv`, which is responsible for task switching in the Zephyr operating system, can be considered as such a function. Ghidra decompiles the function as shown

in Listing 20, which shows that various registers are accessed without having a single P-Code reference to these registers. In this specific case it is possible to generate inline assembly to generate such a register access, but all instructions that are used to restore the context are not captured at all in high-level P-Code and therefore also can't be seen in the decompiled C code.

```
uVar4 = getProcessStackPointer();
*(undefined4 *)(UINT_20000398 + 0x30) = unaff_r4;
*(undefined4 *)(UINT_20000398 + 0x34) = unaff_r5;
*(undefined4 *)(UINT_20000398 + 0x38) = unaff_r6;
*(undefined4 *)(UINT_20000398 + 0x3c) = unaff_r7;
*(undefined4 *)(UINT_20000398 + 0x40) = unaff_r8;
*(undefined4 *)(UINT_20000398 + 0x44) = unaff_r9;
*(undefined4 *)(UINT_20000398 + 0x48) = unaff_r10;
*(undefined4 *)(UINT_20000398 + 0x4c) = unaff_r11;
*(undefined4 *)(UINT_20000398 + 0x50) = uVar4;
```

Listing 20: Decompiled excerpt from `z_arm_pendsv`

A way to solve this issue is to only view low-level P-Code in the transformation, because Ghidra does not perform any kind of optimization or kind of preprocessing for these operations. Although this ensures that all assembly instructions are captured in the transformation, the process of transforming these instructions to LLVM IR can be seen as more complicated. Not only have these P-Code operations no type attached to it, but also are not structured into a control flow graph. Additionally, these operations will also contain the function prolog and epilog, which means that the function in LLVM has to be marked with the attribute `naked`.

4.11 Patch format

Developing a uniform patch file format for the implemented binary rewriter is not as trivial as it may seem, because implementing such functionality into Ghidra itself may be very beneficial for the actual usage of the binary rewriter, but not entirely possible without modifying parts of Ghidra itself. This is because the P-Code representation that is used for the transformation process is only kept around for a single decompilation pass in Ghidra and therefore cannot be used as a basis for developing such a file format. Alternatively patch files can be used to apply patches to the exported LLVM IR code, but this approach introduces a series of problems, because the exported LLVM IR code can be handled like any other source code file. But the decompiled code in Ghidra cannot only change between releases, but also modifications to variables or functions, such as renaming can already produce a rather different LLVM IR code. Therefore, it is possible that not the whole patch can be applied without encountering any problems that lead to malformed LLVM IR code. Nevertheless, with the help of the Linux utilities `diff`

and `patch`, proof-of-concept patch files were created for the prototype in Chapter 5 to investigate the patchability of the prototype.

CHAPTER 5

Evaluation

For the evaluation of the P-Code to LLVM IR framework that was discussed in Chapter 4 two different approaches were used to determine the correctness and performance of the transformation. The first approach utilizes a custom testing framework that can be used to test the correctness of individual functions for the framework and will be described in Section 5.1. Additionally, also a series of benchmarks have been conducted on a limited number of executables to be able to measure the overhead of the transformation process and provide an insight on how the performance shifts when compiler flags are used or patches are applied to the transformed binary. These benchmarks and performance measurements can be viewed in Section 5.2. In Section 5.3 illustrates how a vulnerable binary can be patched with the developed prototype and highlights advantages and disadvantages of the chosen methods. Section 5.4 discusses limitation of the developed prototype that have been discovered in the evaluation.

5.1 Testing framework

The testing framework as shown in Figure 5.1 uses multiple compilation stages to split the compilation and linking part into separate processes. Which also means that the testing framework can be pre-compiled for a series of tests, while the actual functions that are tested are allowed to change. The functions that should be tested will be compiled into an ELF binary without any standard library (`-nostdlib`) and an empty `_start` function. The resulting ELF binary cannot be considered runnable as it will not produce any meaningful output or just crash, because no exit function has been called. In the next step the binary is loaded into Ghidra and exported by the binary rewriting framework to produce a LLVM IR file. Since this framework is only build to test the correctness, no further modifications are made to the produced code to alter the control flow or the behavior of the resulting functions. After that the LLVM IR code is compiled and linked

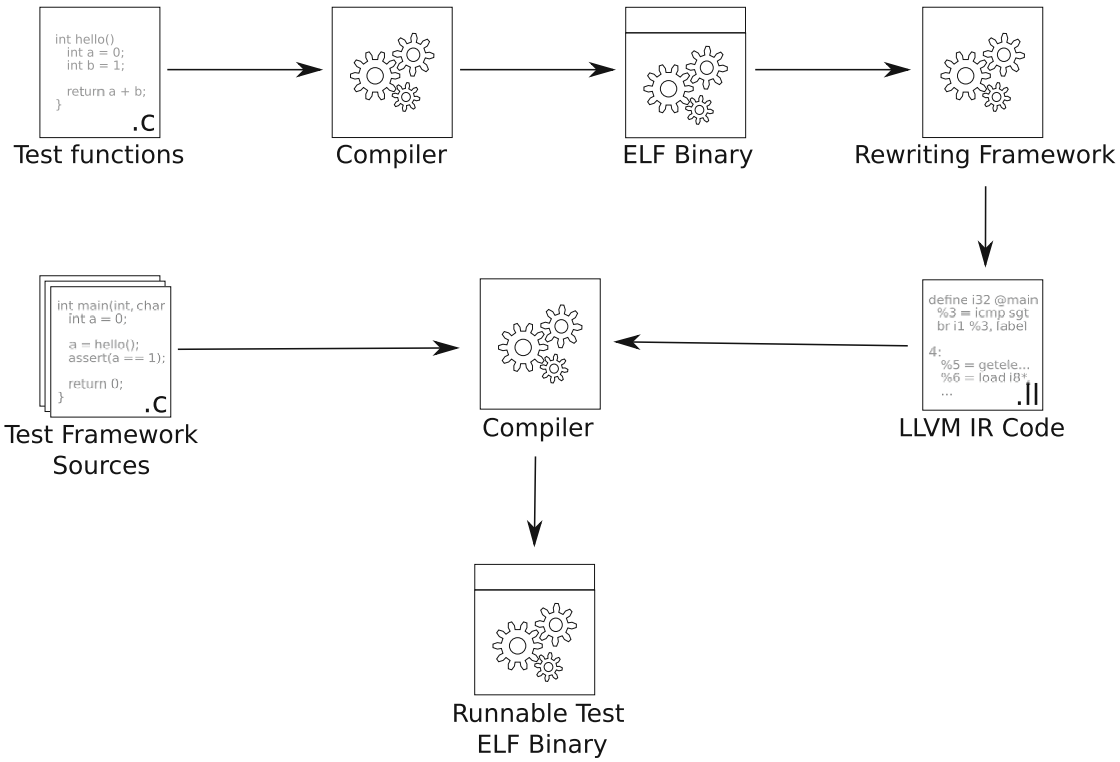

Figure 5.1: Testing Framework

together with the rest of the testing framework to produce a runnable ELF executable to test the functions.

This approach of generating test functions for the rewriting framework has the advantage that the rewriting framework only needs to process minimal assembly code and therefore different scenarios can be created that can be used to evaluate the correctness of the rewriting process for specific cases. Such test functions can be written by hand, extracted from code that is known to causes problems in the transformation process or generated by a code generator that is able to generate standard C code. For an iterative development process this functionality can also be considered to be important, because not only can tests be developed while the prototype of the binary rewriting framework is evolving, but also contribute to identifying breaking changes while still in development. An important aspect of these tests is also that they are not limited to a single optimization level and therefore also the same functions can be exported with different compiler optimization to the binary rewriting framework, which can help identifying problematic parts of the functions. But not all functions may be self-contained, and may interact with other functions such as `printf` or `scanf`, which are not linked when the binary is compiled with the compiler option `-nostdlib`. To solve this problem the functions can be mocked as shown in Listing 21. While this solution is certainly not the best for this case, it solves

the problems with external symbols, which must be handled differently as described in Section 4.9. Such function stubs must then be replaced with external function references, such that the linker can link the correct function in the final testing binary. Also, the compiler flags have to be adapted when the testing functions are compiled, because not only `-nostdlib` is needed, but also `-DBUILD_GHIDRA_BINARY` and `-e__entry__` must be present to signal the compiler that the mocked functions should be used and an empty entry function for the ELF binary should be produced.

```
#ifdef BUILD_GHIDRA_BINARY
#define NO_INLINE __attribute__((noinline))

NO_INLINE void __entry__() { };

NO_INLINE int scanf(const char * format, ...) {};
NO_INLINE int printf(const char * format, ...) {};

#else
#include <stdio.h>
#endif
```

Listing 21: Mocking external functions

5.2 Benchmarks

In this Section multiple benchmarks are used to determine the overhead that can be produced from the transformation from P-Code to LLVM IR. Depending on the assumptions and the optimization level of the compiler, differences should be expected, especially when the compiler can infer much more information from the original source then from the generated code. All Benchmarks are conducted on an Intel(R) Core(TM) i5-4210H CPU with 16GB of RAM and utilize QEMU version 6.1.0 to run ARM binaries on a x86_64 CPU. To gather measurements from different runs the perf utility version 5.14.g7d2a07b76933 was used. Each of the original binaries (compiled from the source code) and the exported binaries have been run multiple times with the following command `perf stat -r 10 -ddd qemu-arm -L /usr/arm-linux-gnueabihf/ <program>`. For compiling the original binaries gcc 11.1.0 und clang 12.0.1 were used, while for compiling the exported only clang was used, because gcc has no support for LLVM IR code as input.

To measure the overhead of the transformation process multiple simple benchmarks where chosen that fulfill the following properties:

- **Tuneable**: A benchmark must be configurable at runtime to change the length or the intensity of the run. For gathering measurements, it is important to take into account that measurements of the whole QEMU process will be collected and therefore it is important to choose a suitable length of each individual run to avoid

that setup and teardown of the virtual environment dominate the results of the run.

- **Reproducible**: A benchmark must be able to produce reproducible results depending on the input parameters. For ensuring correctness of the transformation this is also an important property and therefore such a benchmark can also be used in the form of the testing framework, which was described in Section 5.1.

- **Simple**: A benchmark must be rather simple to avoid running into limitations of the LLVM IR exporter and the binary analysis of Ghidra. Because more complex executables have to be corrected by either editing the exported code or changing the analyzed functions in Ghidra. Although some limitations of the prototype where already mentioned throughout Chapter 4, some concrete limitations will also be discussed in Section 5.4.

5.2.1 fasta

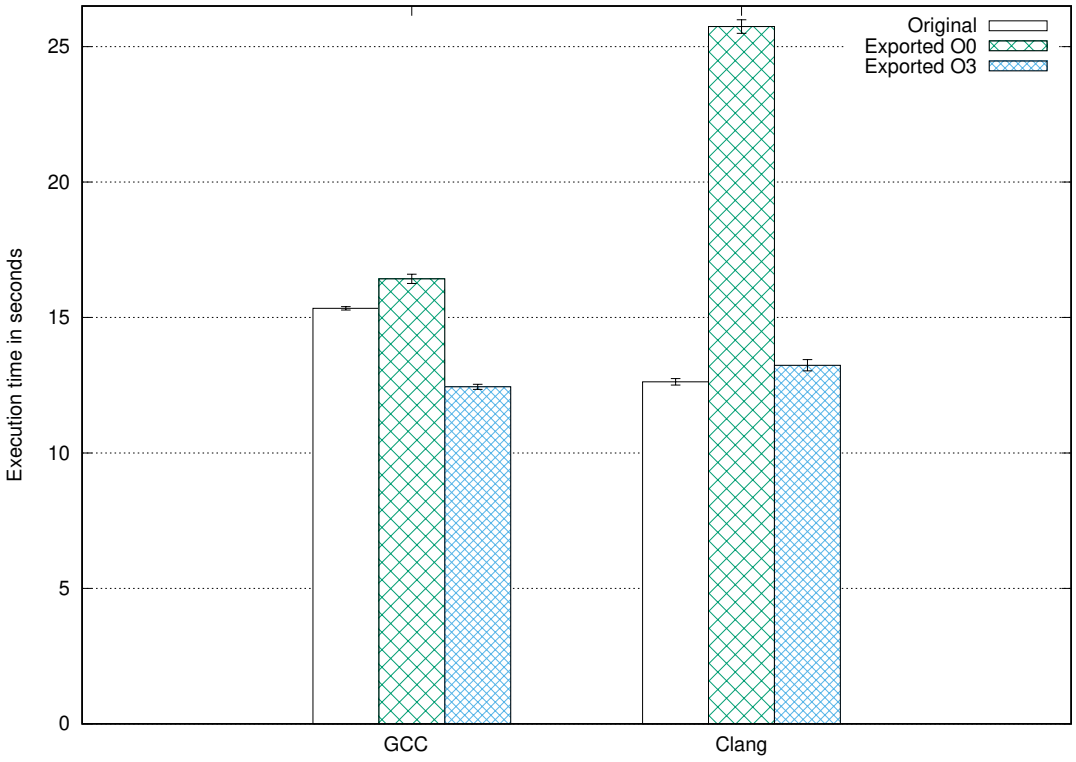

Figure 5.2: fasta benchmark with 250000000 iterations

Shown in Figure 5.2, an implementation of the fasta algorithm from the Benchmarking

Game[1] was used as benchmark. This benchmark mainly uses integer operations with a combination of buffers and memory copies to compute the output. As input parameter 250000000 was used to fix the runtime length of the benchmark to a suitable length. In this Figure we can see that the overall overhead of the transformation is only really relevant if no compiler optimizations are used (using -O0 as compiler argument). Higher optimization levels when compiling the exported binary yields better results placing its runtime very close to the runtime of the original binary. Although the original gcc binary is a little bit slower than the original clang binary, both exports that have been compiled with O3 produce binaries that have similar performance as the original binary file.

5.2.2 mandelbrot

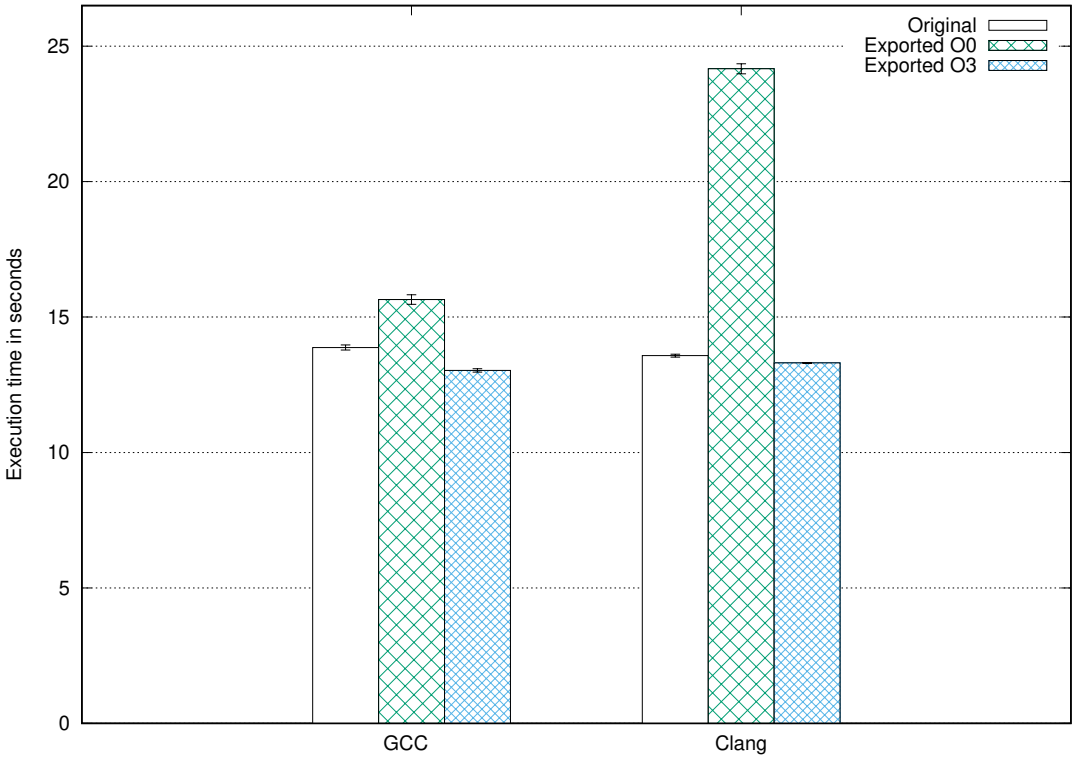

Figure 5.3: mandelbrot benchmark with image size of 2700px

Figure 5.3 shows the collected metrics of another benchmark from the Benchmarking Game[2]. The mandelbrot benchmark mainly utilizes floating-point calculations to generate a Mandelbrot image with the size of 2700 x 2700 pixel. Compared to the fasta Benchmark

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/fasta-gcc-9.html

[2] https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/mandelbrot-gcc-2.html

not much extra memory is needed and the image is generated in a nested loop. In this comparison a similar picture can be observed where the unoptimized builds of the transformed code is slower than the original binary, but the optimized builds (using -O3 as compiler argument) yield similar performance as the original binary.

5.2.3 quicksort

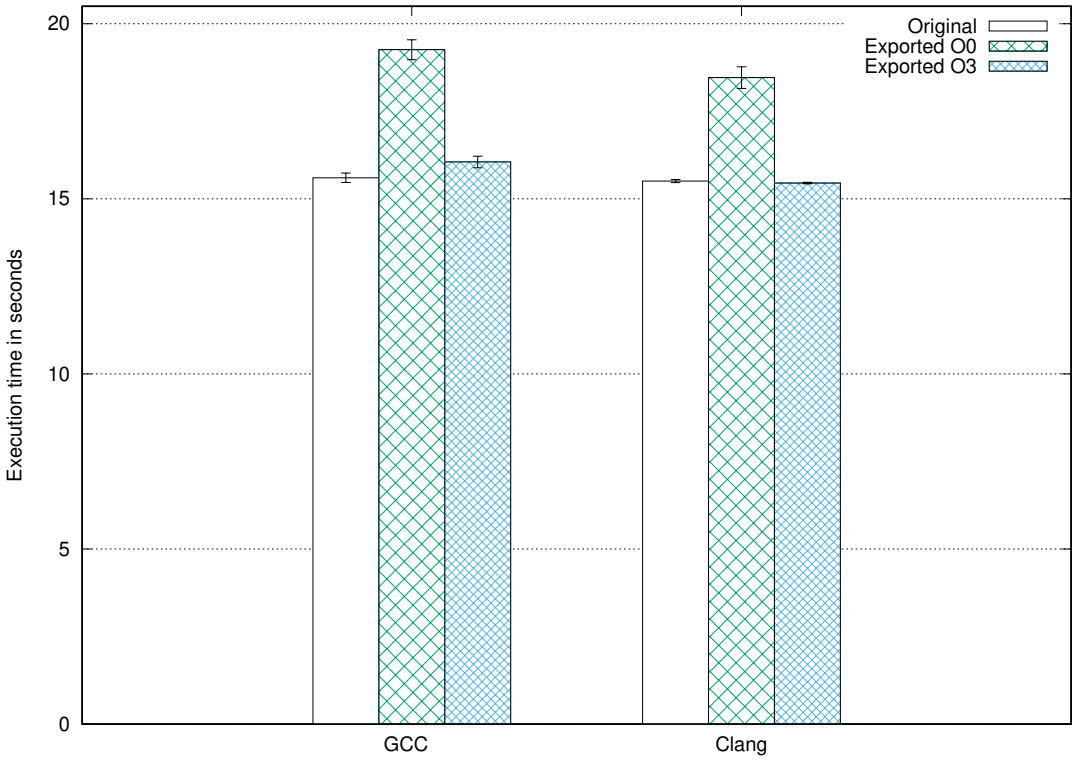

Figure 5.4: quicksort with 12500000 integers

The Figure 5.4 visualizes the performance metrics of the quicksort[3] algorithm. 12500000 integers were generated via a linear congruential generator that is shown in Listing 22 to ensure that each run of the binary the same numbers will be generated and a deterministic output can be expected from the application. Because the quicksort benchmark is much more memory bound then the other benchmarks, the differences between optimized and unoptimized code is not as much as it is the case with the other benchmarks. However, similar to the other benchmarks, also the results of the transformed quicksort are very similar to the original binaries.

[3] https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#C

```
#define M      2147483647
#define A           16807
#define C               0
#define SEED   4242424242
static uint32_t seed = SEED;
#define rand() (seed = (A * seed + C ) % M)
```

Listing 22: Random number generator used for quicksort

5.3 Patching

Since not only the performance of the resulting binary is important to a binary rewriter, but also a sound process of introducing changes into the binary, this section highlights possible ways to utilize the developed prototype to fix the buffer overflow vulnerability shown in Listing 23. In the shown program the function `vuln_function` uses `strcpy` to copy the contents of a string parameter to a local buffer, which has a fixed size. If the string contains more then `BUFFER_SIZE - 1` characters not only other local variables, but also the return address can be overwritten by the user. In most cases this will lead to an abrupt termination of the process by either a segmentation fault or the detection of stack smashing, if a stack canary was present, but it can be exploited by providing a valid function address for the return address. In order to solve this issue it would be necessary

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

#define BUFFER_SIZE 12

void vuln_function(char *s) {
    char buffer[BUFFER_SIZE];
    size_t len = strlen(s);
    strcpy(buffer, s);
    printf("%ld: %s\n", len, buffer);
}

int main(int argc, char *argv[]) {
    if(argc > 1) {
        vuln_function(argv[1]);
        return 0;
    } else {
        return 1;
    }
}
```

Listing 23: Vulnerable program

to rewrite the call to the function `strcpy` such that not more then `BUFFER_SIZE` bytes will be written to the buffer that resides on the stack. But also the introduction of a stack canary can be seen as mitigation, which is described in the following sections.

5.3.1 Stack canary

Utilizing the LLVM compiler toolchain for producing binaries makes it not only easier to apply optimizations, but also greatly simplifies the possibilities to harden the binary post-compilation. For hardening the stack, in the vulnerable program, shown in Listing 23, the attributes of the vulnerable function can be changed to the attributes shown in Listing 24. By using **`sspreq`** clang will insert a stack protector into the function and to avoid inlining of the function **`noinline`** can also be specified. While addition does not fix the buffer overflow vulnerability it can prevent easy exploitation, because before the function returns the stack protector will be checked and if it was modified the program will be terminated. Besides forcing a stack canary on a specific function to prevent easy exploitation, it is also possible to overwrite the attribute of all functions with **`ssp`**, which will only add a stack canary to a function if it is needed. But since the stack is handled as a single big array, it is most likely the case that such a stack smashing protector will be inserted if multiple variables reside on the stack, which will potentially harm the overall performance of the program.

```
attributes #1 = { nounwind sspreq noinline }
```

Listing 24: Stack canary attributes for `vuln_function`

5.3.2 Fixing buffer overflow

But fixing the buffer overflow is not as trivial as changing a few attributes for a function, because the function `strcpy` has to be replaced with `strncpy`, which includes the maximum number of bytes that should be copied to the destination address. In the vulnerable program this can easily be achieved by simply replacing the declaration of the function, because it was build dynamically and `strcpy` is only used in the `vuln_function` function. Additionally, the parameters of the function must also be corrected, otherwise a compiler error will be thrown. In this case the constant 12 is used, since `BUFFER_SIZE` was defined as such and not more space is available on the stack. But it is also possible to expand the stack region arbitrary and include additional logic for handling strings that are too long for the function. Listing 25 shows such an patchfile that can be used to patch the LLVM IR code of the vulnerable program. Alternatively, also an appropriate length check could be used to prevent copying more bytes to the stack, but that would require more complex modifications the the overall control flow structure, as an if statement introduces additional basic blocks or introducing a function call to `@llvm.smin.i32` and replacing the function call to `strcpy` with a looping construct to only copy the required bytes to the buffer.

```
11c11
< declare i8* @strcpy(i8*, i8*)
---
> declare i8* @strncpy(i8*, i8*, i32)
44c44
<   call i8* @strcpy(i8* %buffer, i8* %param_1)
---
>   call i8* @strncpy(i8* %buffer, i8* %param_1, i32 12)
```

Listing 25: Patchfile for the vulnerable program

5.4 Limitations

While the prototype that was described in Chapter 4 can produce acceptable for various binaries, this implement cannot be considered complete or flawless, because the prototype is restricted to binaries that have been compiled from C code. If any features of other languages are used, such as C++ with exceptions, the transformed binaries will not work correctly, as these aspects will not produce a sound transformation. Since the prototype, heavily relies on the correct identification of types, functions and data regions following aspects of the decompiled code can lead to errors while exporting or an incorrect transformation:

Undiscovered Functions: Ghidra utilizes heuristics to be able to find and identify functions. For normal binaries the identification of these usually works reasonably well and missing functions can be promoted from mere labels to functions in Ghidra. But the more complex the binary gets, or the more data structures to indirectly call functions are utilized, the harder it is for Ghidra to fully discover all functions without problems or manual interference. Although the described prototype is capable of processing embedded system images, the correctness of the exported image can vary, because the prototype depends highly on correctly identified functions and types as shown in Figure 5.5. However, as described in Section 5.4.2 exporting non trivial embedded system images it not possible without manually correcting types and function boundaries.

Misidentified stack storage: In Ghidra varnodes have their storage location attached to them, which are utilized by the implement to generate different access patterns when transforming P-Code to LLVM IR. Since the access pattern is different from a normal variable that is stored in a register and the stack, a wrong classification can lead to various errors. For example, sometimes Ghidra identifies a function with a stack size of 0 bytes, but will still generate varnodes that reference the stack region. As the C code in Listing 26 tires to visualize, such cases can occur if a function requires an output parameter, but the value is never accessed in the function. Although the access to the variable in P-Code is correct, the stack frame does not reflect such cases. Such an example leads to errors while transformiwithng the image, because as described in Section 4.6, the

```
                        PTR_LAB_00001448+1_0000420c
0000420c 49 14 00 00    addr        LAB_00001448+1
00004210 d5 14 00 00    addr        LAB_000014d4+1
00004214 ab 3a 00 00    addr        LAB_00003aaa+1
00004218 bf 3a 00 00    addr        LAB_00003abe+1
0000421c 3d 13 00 00    addr        LAB_0000133c+1
00004220 11 13 00 00    addr        LAB_00001310+1
00004224 00             ??          00h
00004225 00             ??          00h
00004226 00             ??          00h
00004227 00             ??          00h
```

Figure 5.5: Ghidra: Undiscovered functions in the Zephyr image

prototype explicitly generates a storage array for all stack operations at the beginning of the function.

```
int error = 0;

// Must be allocated at stack, but will
// not show up in the stack frame
unsigned long flags = 0;

do {
    error = f(&flags);
} while (error != 0);
```

Listing 26: C code example of function that will have a misidentified stack frame

Unreferenced registers: Another limitation of the process are unreferenced registers, which cannot only affect normal registers in the CPU, but also special registers that are located at the FPU, as shown in Figure 5.6. In such cases the prototype does not proceed with the transformation, because these errors cannot be handled correctly without identifying the source of the problem. It could be that the function call uses a different calling convention than recognized by Ghidra, or some operations where not correctly processed, which lead to an unreferenced register in high-level P-Code. Especially floating-point operations can be troublesome in Ghidra 10.0, because comparisons will not correctly be decompiled[4] and potentially other bugs can hinder the construction of correct high-level P-Code. This behavior can be considered wrong when handling kernel functions that are responsible to saving and restoring the userspace context and therefore the discussed implementation does implement the transformation of these functions via low level P-Code. While falling back to low-level P-Code does certainly not solve the overall problem of handling unreferenced registers, it enables the transformation of

[4] https://github.com/NationalSecurityAgency/ghidra/issues/3446

these context switching functions without introducing heuristics that try to identify the problem.

```
dVar12 = (*(double *)(y + iVar8) - *(double *)(y + iVar1)) *
         (*(double *)(y + iVar8) - *(double *)(y + iVar1)) +
         (*(double *)(x + iVar8) - *(double *)(x + iVar1)) *
         (*(double *)(x + iVar8) - *(double *)(x + iVar1)) +
         (*(double *)(&z + iVar8) - *(double *)(&z + iVar1)) *
         (*(double *)(&z + iVar8) - *(double *)(&z + iVar1));
if (dVar12 < 0.0) {
  dVar12 = sqrt(unaff_d8);
}
else {
  dVar12 = SQRT(dVar12);
}
```

Figure 5.6: Ghidra: Unreferenced register

5.4.1 Linking

Linking can be a complex task if additional sections or a custom layout is needed as it is the case with embedded system images. This issue is not fully addressed in this work, and therefore embedded system images that can be exported to LLVM IR may not compile to an executable without making additional changes to the toolchain. One such modification to the toolchain would be the usage of linker scripts that place code and data from different sections into different parts of a system image to conform to the specification of an architecture. When exporting images with the described implementation, it may already be necessary to define such a layout in Ghidra, such that the analysis and the export functionality work, and an automatic export of this information may be possible. But creating linker scripts for different embedded system images is not as trivial, a little bit more than just mapping the position of the sections is needed. However, with the knowledge of how the embedded system image is structured these linker scripts can be used with the compiler to create a custom toolchain for complex tasks. In Listing 27 a trivial example of a linker script is shown, which can be used to structure the layout to conform to the specification of underlying hardware. In order to use linker scripts in combination with the exported LLVM IR code, the LLVM IR code needs to compiled via `llc` into an object file, which can then be used with a linker for the target platform to create an elf binary or embedded system image.

5.4.2 Zephyr / FreeRTOS

Besides the userspace programs shown in Section 5.2 also simple embedded system images with Zephyr 2.5.0-2235 and FreeRTOS 202107.00 were used to test the prototype. Because the prototype relies in many aspects of correctly identified types and functions, the ELF binary with debugging symbols was imported into Ghidra. Although, the presence of debugging symbols means that Ghidra can create a better type mapping and identify

```
MEMORY
{
    ROM (rx ) : ORIGIN = 0x00000000, LENGTH = 256K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}

SECTIONS
{
    .text :
    {
        . = 0x0;
        KEEP(*(.irq_vector_table*))
        *(.text*)
        *(.rodata*)
    } > ROM

    .bss :
    {
        *(.bss*)
        *(COMMON)
    } > RAM
}
```

Listing 27: Example linker script for an embedded system

functions overall better then without it, these embedded system images are far more complex then userspace binaries and therefore other problems can arise. For example, when analyzing the Zephyr image, Ghidra will generate some unions with a size of 0 bytes, while leaving additional unidentified bytes after such a field in a struct. In case of `anon_union.conflict404_for_field_0`, Ghidra could not successfully parse the debugging information and generated such a field. Since such unions do not carry any information which can be used by the prototype to determine why such a union was generated, the default behavior of the prototype is to halt the transformation when such a structure is encountered and notify the user of the occurrence. The user then has to manually resolve such conflicts, otherwise it is not possible to correctly transform the whole data structure to LLVM IR without using any kind of heuristics to determine the correct sizes and ordering of the contained data. Additionally, also other tasks have to performed by the user to ensure that the transformation process succeeds. One such task can be the manual correction of non-returning functions and unreachable code locations. Such unreachable locations can be found throughout highly optimized or operating system code. Figure 5.7 can be seen as an example for such a function. This function is used in FreeRTOS for preparing the environment and starting the first task.

Since this is done by causing a software interrupt, no explicit return instruction is needed as the control flow will never enter the function again. Because of the missing return instruction Ghidra is unable to detect the correct function boundaries. Depending on the layout of the functions, it is possible that the data behind such a function can be interpreted as valid assembly instructions or simply another function starts, which means that a transformation can also succeed if no reference error occur withing the generated P-Code. But if the wrongly interpreted data results in odd assembly instructions the transformation process can fail. In the example that is shown in Figure 5.7, Ghidra is able to detect a malformed assembly instruction, but will nevertheless generate high-level P-Code if the data after the instructions can be interpreted as valid assembly instructions. In this case Ghidra will generate `coprocessor_store(0, in_cr14, unaff_r8);` in C pseudocode before appending the function body of the following function. The prototype will then abort the transformation process, because two unreferenced registers (`in_cr14` and `unaff_r8`) are encountered in the transformation process.

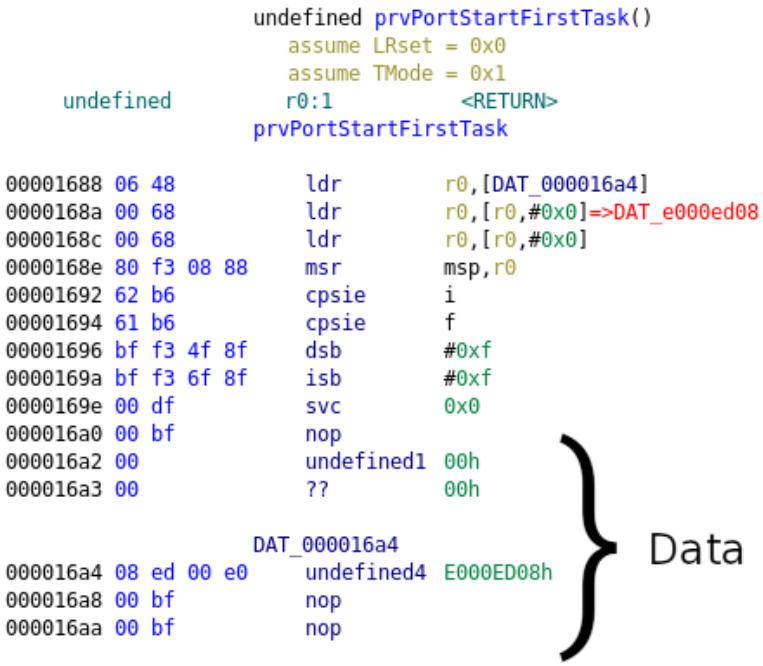

Figure 5.7: Ghidra: Non-returning function

Nevertheless, also more basic issues can occur that can lead later to runtime errors after the exported image has been compiled. Besides misidentification of constant values, for example using a pointer instead of a constant value, it so also possible that specific global variables have to be placed at a fixed memory location. If that is not the case, because these variables are not placed in this location by a linker script, the embedded system can corrupt its state at runtime. Such global variables can be the kernel stack, an additional table of interrupt vectors or just any data structure that is assumed to

have a fixed location in memory. Therefore, it is often necessary to create additional sections that can be used by a linker script to customize the layout of the resulting binary such that the embedded system image can more closely represent the original image. But even if the original linker script is known, which was used to build the embedded system image, it is not entirely possible to use the script for recompiling the LLVM IR code, because not all global variables or even functions may be in the right section. But, it is possible to add this kind of information in Ghidra by splitting or appending new memory blocks, as these blocks attributes will be used in the transformation process to mark the respective section of a global variable or function. For example, with these blocks it is possible to specifiy memory segements that have a special meaning, such as hardware specific addresses. In case of such address ranges, these have to be marked volatile, because a read/write operation of such an address may have a non observable side effect for the compiler, such as writing data to an UART interface oder changing pixels on an attached LCD display. Since only simple embedded system images where processed, only one the peripheral region (`0x40000000 - 0x5FFFFFFF`) [7] is treated by the prototpye on such a way, leading to transformation errors if similar behavior is required for addresses outside of this memory space.

CHAPTER 6

Conclusion

While at first glance utilizing high-level P-Code for the transformation process seems like an optimal solution, because high-level P-Code and LLVM IR share similar constructs for structuring the control flow of the program, it is not entirely the case. Due to different levels of abstractions, often lower-level representations, such as low-level P-Code or assembly instructions have to be minded in the transformation process. In other cases, functionality of atomic instructions may not correctly be represented in high-level P-Code and have to be minded, as described in Section 4.8.3. Therefore, a naive implementation is often not enough to handle all binaries with different levels of optimizations, especially if also the presence or absence of symbols is taken into account. While it may seem like a minor detail that such symbols are present, they can drastically change the quality of the decompiled code of any binary or embedded system image. Usually, the presence of debugging symbols will greatly increase the quality of the generated code, because types are closer to their original counterparts and symbols names for functions, variables and structures are known. But the type conversion, as described in Section 4.1 is not a trivial task and many errors can arise due to the need of implicit type casts. An example for this would be the String data type that describes itself as fixed length string data type, which can be used freely in Ghidra in place of pointers to a type that as the same size as a character. Often such access is also possible without an explicit type cast in Ghidra, which means that the generated code must either be corrected by hand or the prototype has to generate the proper implicit casts. Although the prototype tries to catch as much cases of these implicit casts, not all cases are handled correctly, especially if more complex structures such as multidimensional arrays are accessed. Furthermore, the simplified view of the stack can be seen as problematic, because it does introduce a lot of implicitly generated type casts and also prevents the compiler from optimizing the stack layout of the function. However, without this simplified view the prototype cannot make sure that variables that are stored on the stack will also be stored on the stack in the exported binary, as it would be the case for stack canaries. By enforcing the

storage type of all varnodes in the transformation process problematic stack operations, which require all varnodes to be in a specific order on the stack, can also be handled more easily that it would be the case without the simplified representation. Although, a stack canary can be considered a special case, as it would most likely be inserted by the compiler based on heuristics and compiler flags, not preserving security features in a binary can have negative consequences, because the recompilation process has to make sure to include them or otherwise new vulnerabilities can occur.

For less complex binaries, the approach discussed in Chapter 4 seems to be suitable, as the overall performance overhead of a basic transformation is nearly nonexistent for the benchmarks as shown in Figure 6.1. In some cases, the recompiled binaries can even be a little bit faster than the original ones. But in reality, such cases would be restricted to binaries that have not be compiled with aggressive compiler optimizations, while their transformed counterpart was compiled in such a way to increase the performance. Which means that even without modifications the binary rewriter approach can be used to optimize binaries post compilation. It should be noted that the naive implementation is not able to generate fully platform independent LLVM IR code and therefore compiler flags that are used to optimize the binary for a specific architecture, or limits the instructions that can be used in the compilation, may result in compile time errors.

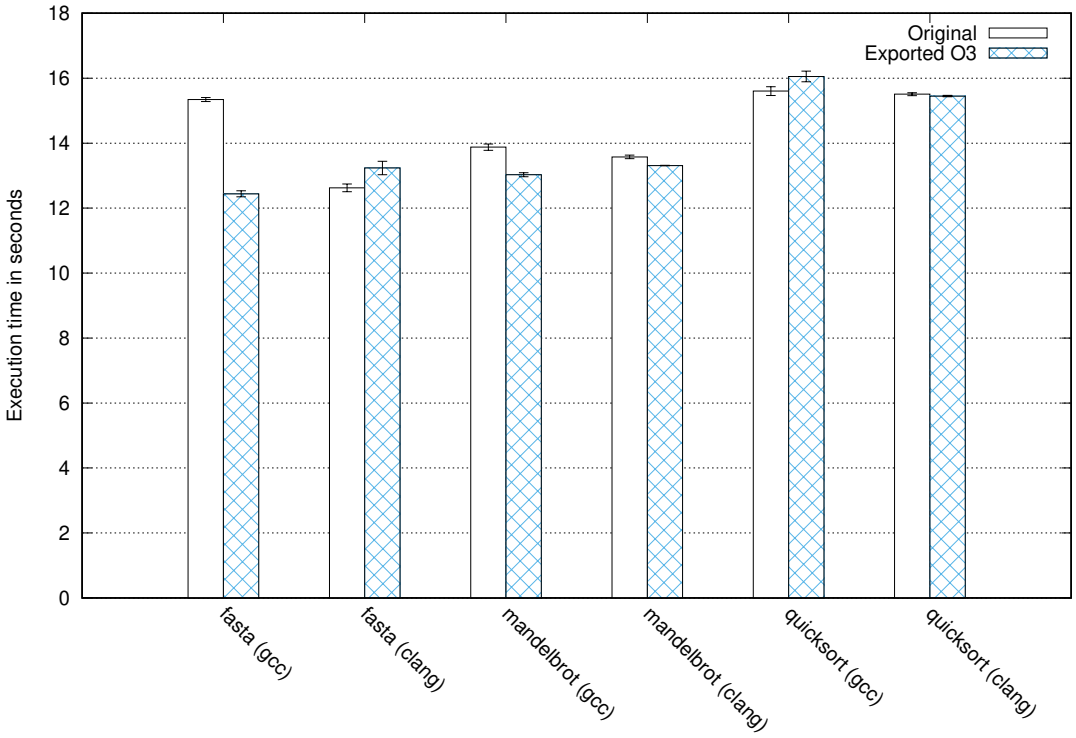

Figure 6.1: Summary of all benchmarks

Although, the implementation of the binary rewriter minds various properties of code that is used in embedded systems, it is currently not possible to use the binary rewriter to process all kinds of images without manually correcting certain aspects in Ghidra. For example, when working with functions that have been highly optimized by the compiler and contain unreachable paths, it is very likely that Ghidra is unable to recover that information and therefore is not able to identify the bounds of the function correctly. Which means that such a function may contain instructions that lead to a corrupted state, or simply the control flow does not mirror the original behavior anymore. With a fitting algorithm or heuristic such problems can also be handled by the analysis step in Ghidra and therefore it is not necessary that the prototype provides this functionality for the export to work in future applications. Nevertheless, it would be beneficial for the exporter to implement and utilize such heuristics to further improve the transformation process.

Another disadvantage of the discussed binary rewriter approach is the distribution of applyable patches and the development of them, because the exported LLVM IR source code may not be reproducible between different versions of Ghidra. Therefore, also these patches cannot be applied to the exported LLVM IR source without manually checking the soundness of the result. One way to ensure that the exported LLVM IR file depends less on the analysis of the binary in Ghidra, would be to ignore all names in Ghidra and only export variables, functions and other named constructs with a deterministic generated name. For normal variables this could be completely omitted, as the LLVM IR framework can automatically name these. Functions on the other hand can be referred to by their address from a standardized offset. However, generating such a file for the export will most likely yield LLVM IR code that is much harder to work with, because the identifiers of variables, which can be used to correlate the variable with the decompiled output in Ghidra, are named differently. Especially when the overall source code size of the exported file is considered. Even for simple and rather small programs, as evaluated in Chapter 5, the exporter produced files with more than 8 KByte of LLVM IR code, while exports from embedded system images had multiple MBytes. Another disadvantage of this approach is also that not only knowledge about low-level structures and assembly code is required, but also knowledge about LLVM IR. But if only minimal changes are needed to improve the overall security of the binary, such as introducing a stack protector for various functions in the binary, a patchfile may not necessarily be needed, because only the attributes needed to be changed. Nevertheless as shown in Section 5.3, the introduction of such a stack protector may help to prevent the overall ability of a malicious user to exploit the application, but does not prevent any unexpected behavior, such as termination.

6.1 Future work

While the developed prototype showed clearly the limitations of the proposed transformation process, it also uncovered areas that are not as well researched as one would expect.

Not only topics surrounding Ghidra, but also the overall analysis of embedded system images with handling platform specifics can be considered as such topics.

Ghidra is a recently open-sourced tool, which utilizes P-Code exclusively for analysis and enriches the low-level P-Code with type annotations and access references. But currently there is not much research available which analyses the quality of the disassembled code or the produced P-Code. Especially when converting P-Code to other representations it is important to not only have a rather precise representation of the control flow, but also a close representation of the types that have been used in the original source code. Depending on the transformation, these properties may be important and handled in a special way. For example, P-Code does not have the notion of atomic access build in, therefore to model such behavior, special `CALLOTHER` functions have to be used, or otherwise the notion of atomicity is lost in the transformation process. Therefore, it would be important to have more research material available that not only describes the handling of different P-Code operations in more detail, but also highlights the usage and manifestation of platform specific features, such as memory fences and atomic read/write operations. Depending on the interpretation of P-Code this can also lead to problems when transforming or even emulating P-Code, because a simple compare and set could not be performed atomically as it was intended in the original binary. Not only this problematic, but also the creation of a useable P-Code emulator can interesting topics for further research.

Besides these issues also the correctness of Ghidra has to be considered, because of its open-source development model it is easy to integrate new features or fix existing bugs when they are encountered, or report it to the community, such that it can be fixed later on. Which also means that Ghidra can evolve a lot over a short period of time and therefore also frequent reevaluation of existing sources may be needed, especially if existing works reveal different shortcomings of the reverse engineering framework. Ghidra supports multiple different loaders for various file formats and instruction sets, which can already help the overall binary analysis process, because often additional meta data can be extracted by these loaders, as they are often more aware of the overall layout then it is the case with P-Code. For example, in case of the raw image loaders for ARM, the interrupt vector table can often be identified by the loader itself if the correct CPU architecture is provided. But the ELF loader does not support this feature, as it is very unlikely that an assembly code inside an ELF binary is intended for the use in an embedded system. Another important topic that has been discussed in this thesis is the type detection and conversion. The developed prototype heavily depends on these detected types and a transformation process can result in wrong results if types are not detected correctly. Especially if the memory where the variable is stored resides on the stack region, it is possible that the recompilation process may rearange the stack layout, which may break the control flow of the program if any reference to this stack region is passed to other functions. Because of these issues, these topics can be used for future research to not only improve general type recognition, but also to develop algorithms for optimizing the stack in the transformation.

Furthermore, this work also raises the question, if the discussed transformation process can be used for compiling transformed for different CPU architectures, because both P-Code and LLVM IR are architecture agnostic languages that may reference architecture specific functions. In such a transformation `CALLOTHER`, atomic operations and other architecture specific access pattern have to be handled with care, such that they have the same effect on the target architecture.

List of Figures

List of Tables

List of Listings

List of Algorithms

Bibliography

[1] H. Böck, "Telekom-routerausfälle waren nur kollateralschaden," 2016. https://www.golem.de/news/telekom-ausfall-router-botnetz-ueber-luecke-in-fernwartungsinterface-tr-064-1611-124751.html, [Online; Last accessed 08-April-2021].

[2] D. dos Santos, S. Dashevskyi, J. Wetzels, and A. Amri, "How tcp/ip stacks breed critical vulnerabilities in iot, ot and it devices." Blackhat Europe 2020, 2020. https://www.blackhat.com/eu-20/briefings/schedule/index.html#how-embedded-tcpip-stacks-breed-critical-vulnerabilities-21503, [Online, Last accessed 15-March-2021].

[3] I. Adascalitei, "Smartphones and iot security," *Informatica Economica*, vol. 23, pp. 63–75, 06 2019.

[4] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), p. 24–35, Association for Computing Machinery, 2016.

[5] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *ACM Computing Surveys (CSUR)*, vol. 52, pp. 1 – 37, 2019.

[6] P. LAFOSSE and J. WIENS, "Modern binary analysis with ils," 2019. https://binary.ninja/presentations/Modern%20Binary%20Analysis%20with%20ILs%20with%20notes.pdf, [Online; accessed 15-March-2021].

[7] *ARMv7-M Architecture Reference Manual*. https://developer.arm.com/documentation/ddi0403/ee/?lang=en.

[8] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the arm and power relaxed memory models," *Draft available from http://www. cl. cam. ac. uk/˜ pes20/ppc-supplemental/test7. pdf*, 2012.

[9] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pp. 45–54, 2002.

[10] T. Committee, "Tool interface standard (tis) executable and linking format (elf) specification," 1995. http://refspecs.linuxbase.org/elf/elf.pdf, [Online].

[11] F. Nielson and H. R. Nielson, "Interprocedural control flow analysis," 1999. http://www2.imm.dtu.dk/~fnie/Papers/NiNi99icfa.pdf, [Online; last accessed 15-March-2021].

[12] L. H. Newman, "The nsa makes ghidra, a powerful cybersecurity tool, open source," 2019. https://www.wired.com/story/nsa-ghidra-open-source-tool/, [Online; last accessed 08-April-2021].

[13] Ghidra, "What's new in ghidra 10.0," 06 2021. https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/Ghidra_10.0.1_build/Ghidra/Configurations/Public_Release/src/global/docs/WhatsNew.html, [Online; last accessed 28-July-2021].

[14] Ghidra, "Sleigh - a language for rapid processor specification," 09 2017. https://ghidra.re/courses/languages/html/sleigh.html, [Online; last accessed 28-July-2021].

[15] Ghidra, "P-code reference manual," 09 2017. https://ghidra.re/courses/languages/html/pcoderef.html, [Online; last accessed 28-July-2021].

[16] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See http://llvm.cs.uiuc.edu.*

[17] L. Project, "Llvm language reference manual," 07 2021. https://llvm.org/docs/LangRef.html, [Online; last accessed 28-July-2021].

[18] M. J. Van Emmerik, "Static single assignment for decompilation," University of Queensland, 2007.

[19] L. Project, "IntrinsicsARM.td," 05 2021. https://github.com/llvm/llvm-project/blob/d1bbe61d1c96c12f890db7b37435f1dce092cc36/llvm/include/llvm/IR/IntrinsicsARM.td, [Online; last accessed 28-July-2021].

[20] G. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[21] C. Niesler, S. Surminski, and L. Davi, "Hera: Hotpatching of embedded real-time applications," in *Proc. of 28th Network and Distributed System Security Symposium (NDSS 2021)*, Internet Society, 2021.

[22] L. Di Bartolomeo, "Armwrestling: efficient binary rewriting for arm," Master's thesis, ETH Zurich, 2021. https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/474088/2/DiBartolomeo_Luca.pdf, [Online].

[23] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton, "Instruction punning: Lightweight instrumentation for x86-64," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), p. 320–332, Association for Computing Machinery, 2017.

[24] A. Engelke and M. Schulz, "Instrew: Leveraging llvm for high performance dynamic binary instrumentation," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, (New York, NY, USA), p. 172–184, Association for Computing Machinery, 2020.

[25] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz, "Binrec: Dynamic binary lifting and recompilation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[26] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, "Revarm: A platform-agnostic arm binary rewriter for security applications," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, (New York, NY, USA), p. 412–424, Association for Computing Machinery, 2017.

[27] D. Ha, W. Jin, and H. Oh, "Repica: Rewriting position independent code of arm," *IEEE Access*, vol. 6, pp. 50488–50509, 2018.

[28] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *NDSS*, 2018.

[29] L. Korenčik, "Decompiling binaries info LLVM IR using MCSEMA and Dyninst," 2019. https://is.muni.cz/th/pxe1j/thesis.pdf, [Online].

[30] K. Kirchner and S. Rosenthaler, "Bin2llvm: Analysis of binary programs using llvm intermediate representation," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[31] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), p. 295–308, Association for Computing Machinery, 2013.

[32] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1497–1511, IEEE, 05 2020.

[33] M. Ahmadi, P. Kiaei, and N. Emamdoost, "Sn4ke: Practical mutation testing at binary level," 2021.

[34] E. M. Schulte, J. Dorn, A. Flores-Montoya, A. Ballman, and T. Johnson, "Gtirb: Intermediate representation for binaries," *ArXiv*, vol. abs/1907.02859, 2019.

[35] M. Rodler, "Unions," 10 2017. `https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/basic-constructs/unions.html`, [Online; last accessed 28-July-2021].

[36] Ghidra, "JavaDoc - BasicBlockModel." `https://ghidra.re/ghidra_docs/api/ghidra/program/model/block/BasicBlockModel.html`, [Online].

[37] Ghidra, "Additional p-code operations," 09 2017. `https://ghidra.re/courses/languages/html/additionalpcode.html`, [Online; last accessed 28-July-2021].

[38] L. Project, "The often misunderstood gep instruction," 07 2021. `https://llvm.org/docs/GetElementPtr.html`, [Online; last accessed 28-July-2021].

[39] L. Project, "Llvm atomic instructions and concurrency guide," 07 2021. `https://llvm.org/docs/Atomics.html`, [Online; last accessed 28-July-2021].

[40] A. Terekhov, "C/c++11 mappings to processors," 12 2008. `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`, [Online; last accessed 28-July-2021].

[41] M. Hommey, "Bug 46770 - Replace .ctors/.dtors with .init_array/.fini_array on targets supporting them," 12 2010. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=46770`, [Online; last accessed 28-July-2021].

[42] J. D. Anglin, "Bug 19170 - ___gmon_start___ defined in hppa in crtn.S," 10 2015. `https://sourceware.org/bugzilla/show_bug.cgi?id=19170`, [Online; last accessed 28-July-2021].

[43] gtackett, "Ghidra symbol definitions–preserving weak/non-weak property at import #2720," 02 2021. `https://github.com/NationalSecurityAgency/ghidra/issues/2720`, [Online; last accessed 28-July-2021].

[44] T. C. Team, "Clang 13 documentation - attributes in clang," 2021. `https://clang.llvm.org/docs/AttributeReference.html#interrupt-arm`, [Online].

[45] A. F. M. Abdelrazek, "Exception and interrupt handling in arm," 2006. `http://classweb.ece.umd.edu/enee447.S2019/ARM-Documentation/ARM-Interrupts-3.pdf`, [Online].