ADVANCED CODE REWRITING THROUGH
MULTI-PATH BASED BINARY RECOVERY

A Thesis by
Taddeüs Kroes

Supervised by
Herbert Bos, Istvan Haller
and Lucian Cojocar

August 2015
Vrije Universiteit
Amsterdam
ABSTRACT

This work presents a binary rewriting framework that combines lifting of binary code to LLVM IR with symbolic execution. The framework provides a foundation for advanced applications of binary rewriting, integrating previously existing compilation techniques from the LLVM toolchain. Our first contribution is a front-end based on S²E that lifts executable code to LLVM IR, using symbolic execution to cover multiple program paths. The second contribution comprises a transformation engine that removes the effect of virtualization from lifted code. This transformation engine, the most significant part of our work, produces an IR suitable for binary rewriting and subsequent compilation. The third and final back-end component uses the LLVM toolchain to lower the lifted code back into an executable binary. The result is a recovered binary that is functionally equivalent to the input binary, combined with the results of binary rewriting.

We evaluate the accuracy of our framework by successfully recovering a number of input binaries, including several real-world binaries from GNU Coreutils and a binary obfuscated by the state-of-the-art obfuscator CodeVirtualizer. Using symbolic execution, we have recovered code which is representative of 60% of the source code of Coreutils echo, and 100% on two virtualization-obfuscated toy programs.

We have also measured the execution time of recovered code. We obtained an acceptable 76% slowdown on the Dhrystone benchmark, and a 15% speedup on a SHA-512 hashing program. These results are encouraging for future exploration of possible applications of our approach.
ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Herbert Bos, for his guidance and the confidence he placed in me to complete this difficult project. I am also grateful to my daily supervisors Istvan Haller and Lucian Cojocar, whose advice, ideas and practical assistance have been invaluable for the success of the thesis. Furthermore I would like to thank my friends and colleagues at both the VU and UvA universities, for offering a listening ear when needed and making the experience in general a positive one. In particular, I want to thank Raphael Poss for his advice and valuable discussions regarding the thesis report. Final thanks go out to all of the other people that provided support with comments and discussions, including but not limited to Simon Polstra and Alyssa Milburn.
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Dynamic binary code lifting</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Low-level code transformation and analysis with LLVM</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Multi-path exploration</td>
<td>5</td>
</tr>
<tr>
<td>2.3.1 Enhanced dynamic analysis using multi-path exploration</td>
<td>5</td>
</tr>
<tr>
<td>2.3.2 Symbolic and concolic execution</td>
<td>5</td>
</tr>
<tr>
<td>2.3.3 Selective symbolic execution with $S^2E$</td>
<td>6</td>
</tr>
<tr>
<td>3 Framework overview</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Design</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Implementation</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Usage and workflow</td>
<td>10</td>
</tr>
<tr>
<td>4 Capturing code with $S^2E$</td>
<td>11</td>
</tr>
<tr>
<td>4.1 VM setup and program startup</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Relevant code selection</td>
<td>12</td>
</tr>
<tr>
<td>4.2.1 Automatic module creation for input binaries</td>
<td>12</td>
</tr>
<tr>
<td>4.2.2 Filtering library code</td>
<td>13</td>
</tr>
<tr>
<td>4.3 Extracting code from the $S^2E$ program state</td>
<td>13</td>
</tr>
<tr>
<td>4.3.1 Enforcing creation of LLVM IR</td>
<td>14</td>
</tr>
<tr>
<td>4.3.2 Recording basic block successors</td>
<td>15</td>
</tr>
<tr>
<td>4.4 Exercising the binary</td>
<td>16</td>
</tr>
<tr>
<td>5 Transformation of captured code into executable binary</td>
<td>17</td>
</tr>
<tr>
<td>5.1 Removing $S^2E$ runtime dependencies</td>
<td>17</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Binary executables for which only machine code is available, without access to source code, have historically been difficult to analyse and transform. A myriad of research has been performed on binary rewriting. Applications include enforcement of security policies [1, 2], advanced binary analysis [3, 4] and binary optimization [5]. A wide range of binary analysis tools is available to researchers. These tools typically lift executable code to an intermediate representation (IR) in order to perform transformations. Many tools have developed their own custom IR, each incompatible with that of previously existing tools. This makes the lifted code unsuitable to apply a wide range of existing compiler level transformations, from which analysis could greatly benefit. SecondWrite [6], a recently developed framework for static binary rewriting, has uses a compiler level IR as a target for binary code lifting, and has successfully integrated custom analysis with powerful, previously existing compilation techniques.

Target binaries in the field of binary rewriting are often release builds of software, stripped of symbols and debugging information. In malicious applications, headers may even be corrupted and debugging information falsified. Advanced program obfuscators such as packers [7, 8] and virtualization-obfuscators [9, 10] also modify the executable code of a binary in such a way that the original machine code is hidden behind a layer of emulation code, causing analysis tools to analyse this generic emulation code rather than the code of interest. Static analysis of such binaries is well-known to be difficult, and makes assumptions about structure of the executable code in order to identify code. Because developers of binaries can employ simple countermeasures to render static analysis ineffective, dynamic analysis is often used to obtain an understanding of a binary programs by observing its runtime behaviour.

However, a significant drawback of many existing dynamic analysis tools is that they are limited to single-path execution traces, possibly ignoring core behaviour of the analysed software that is only executed when certain input conditions are met. Since the input of black-box binaries such as malware is typically unknown, a more useful approach is one that automatically finds input values that trigger certain behaviour so that multiple code paths are discovered. Well-known multi-path execution techniques include fuzzing [11] and symbolic execution [12, 13]. The latter uses symbolic expressions to represent
values which cause branches in the execution, recording different constraint on the expression for each branch direction. A constraint solver is used to find concrete values for symbols which will result in execution of the corresponding code path. These techniques are commonly grouped in dynamic analysis frameworks to aid researchers investigating malware. Examples are S\textsuperscript{2}E [14] and BAP [15].

In this thesis, we combine binary rewriting with multi-path dynamic analysis. We have implemented a framework on top of existing tools that lifts executable code to the well-known Low-Level Virtual Machine (LLVM) IR in order to perform binary rewriting. The lifted code is lowered back into machine code, producing an executable binary. Our framework integrates symbolic execution to enable multi-path exploration of binary programs. The framework targets x86 binaries on both Linux and Windows.

Chapter 2 introduces some key concepts to aid in understanding this work. Chapter 3 provides a high-level overview of our framework, followed by an in-depth description of implementation details in chapters 4 and 5. Chapter 6 evaluates our framework implementation. Chapter 7 compares our work to other research. Chapter 8 highlights areas for future research. Finally, Chapter 9 formulates our conclusions.
2. BACKGROUND

Our binary recovery framework uses a number of techniques that are explained and motivated in this chapter. The framework makes use of dynamic binary code lifting, code transformation and multi-path exploration.

2.1 Dynamic binary code lifting

Static vs. dynamic disassembly  When lifting machine code from a binary to a higher-level IR, the first key challenge is to identify which bytes of the binary are code. These bytes must be disassembled into machine code which can then be transformed into the desired IR. There are two ways of disassembly machine code: static and dynamic. Static disassembly of binaries is known to be difficult and can be foiled by obfuscation techniques [16, 17, 18]. Reverse engineering and malware researchers therefore often resort to a dynamic approach [19, 20]. When a binary is executed in a virtual environment, the instructions that are executed by the virtual processor can be observed. This way, the code executed by a binary can be determined reliably. Because obfuscated binaries are in the intended target set of software for our framework, we have chosen the dynamic approach to reliably capture the code of such binaries.

Dynamic Binary Translation  Dynamic binary analysis frameworks rely on virtualization techniques to execute binaries in order to have access to executed code and easily sandbox applications in user-mode. For instance, S2E [14] and BAP [15] translate code that is to be executed from the source instruction set into a target instruction set for execution. A Dynamic Binary Translation (DBT) engine injects instrumentation that makes the code operate on virtualized hardware rather than on physical hardware. Code is typically “lifted” by a front-end translator from the source instruction set (e.g., x86) into some intermediate language (e.g., TCG) that allows for easy transformations. The generated intermediate code can be translated into the host instruction set by a back-end translator for native execution, in order to achieve native performance. Alternatively, the IR can be executed by an interpreter to apply advanced analysis techniques such as symbolic execution.
Intermediate code generated by a DBT engine is altered in various ways to run on virtual hardware. Control flow instructions are modified to interleave execution of the emulator and of virtualized software. Memory accesses are changed to access the program’s address space in virtual memory. Side effects of instructions (e.g., flags set by the x86 add instruction) are typically explicitly represented in the IR, which is composed of a simple instructions that do not have side effects. Code 2.1 shows an illustrative example of BIL code (BIL is the IR of BAP) corresponding to an add instruction. The simplicity and explicitness of such an IR makes it an ideal target for analysis and rewriting, since it provides a unified language for implementing transformations on code of any source architecture.

**Code 2.1** BIL code for `add %rax, %rbx`.

```plaintext
addr 0x0 @asm "add %rax,%rbx"
label pc_0x0
T_t1:u64 = R_RBX:u64
T_t2:u64 = R_RAX:u64
R_RBX:u64 = R_RBX:u64 + T_t2:u64
R_CF:bool = R_RBX:u64 < T_t1:u64
R_OF:bool = high:bool((T_t1:u64 ^ T_t2:u64) & (T_t1:u64 ^ R_RBX:u64))
R_AF:bool = 0x10:u64 == (0x10:u64 & (R_RBX:u64 ^ T_t1:u64 ^ T_t2:u64))
R_PF:bool = ^low:bool(let T_acc:u64 := R_RBX:u64 >> 4:u64 ^ R_RBX:u64 in
let T_acc:u64 := T_acc:u64 >> 2:u64 ^ T_acc:u64 in
T_acc:u64 >> 1:u64 ^ T_acc:u64)
R_SF:bool = high:bool(R_RBX:u64)
R_ZF:bool = 0:u64 == R_RBX:u64
```

The availability of existing binary analysis frameworks and low-level intermediate languages eliminates the need of developing a custom IR for our framework. Instead, we have implemented binary recovery on top of an existing framework that uses the widely known LLVM IR.

### 2.2 Low-level code transformation and analysis with LLVM

A widely used IR for instruction-level code analysis is that of the LLVM compilation framework [21]. The LLVM IR consists of a relatively simple instruction set that uses Static Single Assignment form [22] to efficiently implement optimizations. Analysis and transformations in LLVM are structured in “passes”. After conversion of a source language to LLVM IR, an arbitrary number of passes can be used to transform and
optimize the IR before assembly into the target instruction set. The LLVM toolchain allows the user to combine custom passes with powerful built-in optimization and analysis passes. Furthermore, an extensive and well-documented API is defined for including LLVM front-ends, back-ends and transformations in third-party tools, making LLVM an ideal platform for analysis of low-level code [23, 24, 25, 14, 6].

2.3 Multi-path exploration

The main problem of dynamic analysis is the choice of input parameters to pass to the program under analysis. If the input does not meet certain conditions, core program behaviour may be missed. For example, the Michelangelo virus [26] was programmed to only execute its malicious code on a specific day of the year (March 6, Michelangelo’s birthday).

2.3.1 Enhanced dynamic analysis using multi-path exploration

Moser et al. [27] implement a solution to this problem by tainting memory values returned by system calls and identifying branches that depend on these values. Operations on tainted values are tracked and their results are stored as expressions. Because the predicate condition that determines control flow is tainted, one can track the relation between data and code execution. At a branch where the predicate condition contains a tainted value, the corresponding expression is solved using a linear constraint solver to find a value that leads to execution of the alternate direction of the branch. The framework also records a snapshot of the program at the branch, from which it follows the alternate branch direction after the current code path terminates in a depth-first order. Using this method the researchers were able to identify additional code paths in 151 out of 308 real-world malware samples.

2.3.2 Symbolic and concolic execution

Moser’s technique, described in previous section, can be further improved using program testing techniques that aim to automatically find program inputs in order to obtain higher code coverage of a tested program. A well-known technique that has a wide range of applications is symbolic execution [12, 13]. In symbolic execution, code is executed in a simulated environment in which the values of interest are represented by symbols rather than concrete values. Predicate expressions of branch statements that include a symbol are recorded in a list of path constraints, where a different constraint
on the expression is recorded for each branch direction. When a path terminates, the constraint system is solved to find a set of concrete values for symbols (e.g., input parameters) that lead to execution of the path.

Symbolic execution does not scale to large programs. A linear increase in program size may result in an exponential increase in the number of paths to cover. For instance, when if-statements are added at the beginning of a program, the number of program paths grows exponentially. This is referred to as the path explosion problem. Because computation resources are finite, symbolic execution may terminate before all code paths are discovered, yielding a subset of all possible program paths. Solutions the path explosion problem include the application of path-finding heuristics to increase code coverage [28] and parallelization of independent paths to reduce execution time [29]. Symbolic execution frameworks such as KLEE [23] and EXE [30] offer search strategies to prioritize certain paths over others, for example targeting optimal code coverage or directing execution towards certain target code.

Symbolic execution suffers from low execution speed in comparison to execution with concrete values. Efforts have been made to combine the best of both worlds by executing a program with concrete values, while maintaining path constraints with symbolic values. At a branching point, a constraint solver is used to find a set of concrete values that must be updated in order to follow the other branch direction. This scheme is called concolic execution (a contraction of “concrete” and “symbolic”), of which CUTE [31] is a well-known implementation for C programs.

2.3.3 Selective symbolic execution with S²E

Symbolic execution frameworks are primarily designed to find bugs in software by exploring code paths that are missed by manually crafted test cases. As such, they do not take binary programs as input, but rather source code or an IR constructed directly from source code, allowing for high-level reasoning about symbolic values. For example, KLEE uses an LLVM interpreter and CUTE uses a C-like language constructed from input C code. To perform symbolic execution on a binary, it must be disassembled and the machine code transformed into an IR suitable for the symbolic execution engine of choice. Furthermore, execution of a binary may have undesired side effects on the host system. This is particularly the case for malware binaries which may try to corrupt the system or steal information. Analysis of binary programs is therefore often conducted in a virtual machine.
$S^2E$ is a framework that combines symbolic (and concolic) execution of binary programs with virtualization. It uses QEMU [32] for virtualization and KLEE for symbolic execution. A custom LLVM back-end for the Tiny Code Generator (TCG) of QEMU is used to generate input for the KLEE symbolic executor. The $S^2E$ execution engine is able to switch between symbolic mode and regular DBT mode. Code runs in DBT mode by default, attaining high performance. When a symbolic memory value is used, symbolic mode is engaged, in which the LLVM TCG back-end is used to generate LLVM IR for execution by KLEE. The ability to switch between these execution modes is called Selective Symbolic Execution (abbreviated SSE, hence the name “$S^2E$”).

Selective symbolic execution successfully combines multi-path exploration with dynamic analysis of binary executables. Combined with the use of LLVM IR for code translation, $S^2E$ provides a solid foundation for our binary recovery framework.
3. FRAMEWORK OVERVIEW

3.1 Design

Chapter 2 motivates the use of S²E as the foundation of our framework. S²E offers selective symbolic execution to perform symbolic execution on an input binary. It furthermore lifts machine code to LLVM IR on which code transformations can be performed. The LLVM compiler toolchain is used to lower LLVM IR to a recovered binary.

Figure 3.1 shows the overall design of our framework. The input binary is executed by S²E inside a virtual machine, optionally using symbolic execution to increase code coverage with regards to single-path execution. Executed code that belongs to the input binary is exported as LLVM IR. The captured IR is used as input for optional transformations and the result is compiled back into an executable binary. Transformations are indeed optional; the framework in its basic form can be used to simply capture a binary without transforming its code, for example to obtain a single-path variant of the code.

![Figure 3.1.: Framework design.]

3.2 Implementation

Figure 3.2 shows a detailed view of the implementation of our framework. Custom S²E plugins export LLVM IR for all executed code. Supported binary formats are 32-bit ELF for Linux and 32-bit PE for Windows, for the x86 architecture. Chapter 4 discusses in detail how code is exported from S²E.

The captured IR is used as input for user-defined transformations, after having been transformed by a series of LLVM passes to produce IR suitable for compilation (the Deinstrumentation and Optimization boxes). The compiled IR is paired with a number
Figure 3.2.: Framework implementation. The labels of highlighted areas show our contributions.

of hand-written assembly helpers whose necessity is introduced by some deinstrumentation passes. Chapter 5 discusses these code transformations at length.

The Feature extraction component is new with regards to Figure 3.1. It comprises a number of utility scripts that extract features of the input binary that are not included in the captured IR, for example, the contents of data sections. Chapter 5 describes all cases where the framework relies on such extracted features. Some of the extracted features could in principle be encoded in the captured IR by additional S2E plugins. This is more reliable because binary metadata such as the symbol table and relocation table binary may be obfuscated to foil static inspection. An S2E plugin has the view of program at runtime. This allows memory inspection but also library call tracing. For PE binaries, such an inspection functionality exists within the S2E framework, which we use to export annotate code that calls library functions. For ELF binaries this functionality does not exist, and its development is not in the scope of this thesis. In particular, we use scripts to extract symbols, dynamic relocations, addresses of compiler-generated code and data section contents from ELF binaries.

Because of the S2E limitations with regards to ELF binaries, we have chosen to focus on conventionally compiled binaries of the ELF format. Hand-crafted or obfuscated binaries such as malware are only supported in the PE format. An additional feature, exclusive to ELF binaries, is the removal of compiler-generated setup code. A compiler generates code that sets up program arguments and various other aspects of the runtime environment. We remove this compiler-generated code to obtain a CFG that closely resembles the source code of the input binary. This enables manual inspection and debugging. In this way the human analyst is not distracted by unused code.
3.3 Usage and workflow

Binary recovery is separated into two main tasks. First, capturing code with S²E. Second, transformation of captured code into an executable binary. A collection of command-line utility scripts and GNU Makefiles group each of these tasks in a command, resulting in only two commands to execute in order to run an input binary in the framework and obtain a recovered binary. The following examples demonstrate the ease of use of the framework.

The following snippet recovers the PE binary example.exe. run-vm.sh loads the “winxp” VM at snapshot “example”. We assume that the user has prepared this snapshot with example.exe ready to be executed. Symbolic execution is always enabled (hence the -sym flag), but S²E will not engage symbolic mode until a symbolic memory value is encountered. For reasons explained in Section 4.1, the user needs to specify options for symbolic arguments in a configuration file when dealing with the Windows VM, and interact with the VM in order to begin execution the input binary. S²E produces the output directory “s2e-out-example” containing LLVM bitcode exported by our custom plugins, along with debug information and log files.

$ qemu/run-vm.sh winxp -sym example
$ recover/recover.sh s2e-out-example/ example-recovered

The next snippet recovers a single-path variant of ELF binary example, passing “foo” as the first argument. cmd-debian.sh is a wrapper script that calls run-vm.sh debian cmd, where cmd is the name of a snapshot that loads a binary into the VM at runtime and executes it with the specified command-in arguments (see also Section 4.1).

$ qemu/cmd-debian.sh ./example foo
$ recover/recover.sh s2e-out-example/ example-recovered

Recover a multi-path variant of ELF binary example, passing 0 to 2 arguments consisting of 3 symbolic bytes:

$ qemu/cmd-debian.sh ./example --sym-args 0 2 3
$ recover/recover.sh s2e-out-example/ example-recovered
4. CAPTURING CODE WITH S\textsuperscript{2}E

This chapter describes in detail how we use S\textsuperscript{2}E to lift code to LLVM IR. We discuss the workflow of using virtual machines, selection of code to export, recording of control flow transitions and creation of symbolic inputs.

4.1 VM setup and program startup

The S\textsuperscript{2}E framework contains two QEMU system binaries for each hardware architecture. First, an adapted version of the VM which is used for symbolic execution. This VM has an additional “symbolic mode” on top of the default Dynamic Binary Translation (DBT) mode, in which executed code is translated to LLVM and executed with KLEE. The VM switches between the symbolic and DBT modes depending on whether the executed code depends on symbolic values in memory. This technique is called “selective symbolic execution” [14] (see also Section 2.3.3).

The adapted QEMU VM is considerably slower than vanilla QEMU, making the initial step of configuring a VM tedious and time-consuming. This includes installing and configuring an operating system on a VM, booting the system and setting up the environment for execution of the software under analysis. To speed up this process, S\textsuperscript{2}E offers a vanilla QEMU binary of the same version as the adapted VM which can be used to set up the system up to the point where the targeted software is ready to be executed, optionally using KVM [33] for better performance. At this point, the user saves a snapshot of the VM to disk, exits the VM, and loads the snapshot into the S\textsuperscript{2}E-adapted version of QEMU to commence symbolic execution of the target software.

There are two ways to trigger program execution when continuing execution from a snapshot. The first is to manually start the command in the VM after the snapshot is loaded, which requires the user to wait until the snapshot is loaded. For Linux, there is a second way which does not require user interaction, by means of the \texttt{s2eget} utility. This program utilises custom S\textsuperscript{2}E instructions: instructions outside the instruction set of the target machine used by the adapted VM to trigger events. The particular instruction used by \texttt{s2eget} causes vanilla QEMU to stall so that a snapshot can be saved. In the adapted VM, execution is resumed and a program is copied from the
host into the guest and subsequently executed. With the help of s2eget, we need only a single snapshot to execute any binary with any command-line arguments without having to interact with the VM, boot it and create a new snapshot for each binary (which is the case for Windows binaries).

4.2 Relevant code selection

The framework only exports code belonging to the binary under investigation. To do this, it must be able to distinguish the “relevant” code to export from other code that is executed by S²E. We do this by grouping execution events in so-called “modules” on the basis of the origin of the executed code, allowing us to filter out library code.

4.2.1 Automatic module creation for input binaries

Analysis in S²E works in a modular fashion, by triggering events in plugins which subsequently produce output, influence execution or trigger new events. The core plugin triggers execution events for all executed code, including kernel code and library code. To distinguish between different programs S²E uses the concept of “modules”. A module is defined by a name and a range of memory addresses indicating the code belonging to that module. The ModuleExecutionDetector plugin groups execution events by corresponding modules, and is used to filter events belonging to the program under investigation.

S²E offers some ways to automatically create modules for loaded programs. For Windows images, built-in plugins automatically create modules for binaries and libraries. For Linux images, however, such plugins do not exist. Instead, code selection here relies on a preloaded library¹ called init_env.so that works by replacing _libc_start_main with a function that creates a new module before resuming execution. The memory range for this module is read from /proc/self/maps in the guest system. While this method works reliably for most use cases, it poses a problem when trying to capturing all code belonging to a binary; Any code executed before _libc_start_main is executed before the module is created, causing plugins to miss execution events for this code. In regularly compiled binaries, the missed code comprises only fixed bootstrapping code at the _start symbol which can be solved by adding equivalent code to the exported code. However, reliance on a standard libc bootstrapping code being

¹A preloaded library, specified with the LD_PRELOAD prefix, is loaded before any other library. A preloaded library can overwrite functions from libraries that are loaded later, e.g., libc.
present limits the set of possible target programs. Irregular binaries such as obfuscated malware or manually crafted drivers may contain payload code that is executed before \texttt{\_\_libc\_start\_main}, or not even contain the symbol at all (resulting in no exported code). The implementation of a solution to this problem is not within the scope of this thesis.

### 4.2.2 Filtering library code

Only code from modules registered with the \texttt{ModuleExecutionDetector} plugin will trigger module execution events. Since only one module is registered, all other code, including library code, will only trigger global execution events in the core plugin. This causes a problem for the built-in S\textsuperscript{2}E plugin that detects library calls, which assumes module execution events for libraries. We have developed a custom plugin to detect library calls. The plugin abuses the fact that module execution events are triggered twice (once globally by the core plugin and once by the \texttt{ModuleExecutionDetector} plugin) whereas events for other code are only triggered once (only by the core plugin). By comparing these sets, transitions into and from library modules can be reliably detected.

Note that the plugin does not know which library function is called, only that a library function is called. This information is, however, necessary to correctly reconstruct library calls. Section 5.2.1 explains how symbol information is used to annotate a module transition with the name of the library function that is called.

**Dynamically extracted code**  Dynamically extracted code, such as the payload code for packers [7, 8], is extracted during the execution of the program and therefore not yet visible during program initialization. Such code will therefore be seen as library code and not be exported. Currently, our framework is not able to correctly handle dynamically extracted code. A possible solution to this problem could be to register dynamically allocated memory to the module under investigation by continuously monitoring changes to /\texttt{proc/self/maps}.

### 4.3 Extracting code from the S\textsuperscript{2}E program state

Given execution events that are grouped by module, all that is left is to export the LLVM IR for the module under investigation. To do this it is necessary to understand how S\textsuperscript{2}E manages code under the hood. Execution is divided into execution states, where each state represents a particular code path. When the symbolic executor forks at a branching point, it creates a new state for the new discovered code path. The
execution state is passed as an argument to all event handlers, and as such provides an API for plugin developers. The execution state contains code at the granularity of a basic block\(^2\), which is the unit of execution in the KLEE symbolic executor. When an execution event is triggered at program counter \(X\), the execution state contains a pointer to the basic block starting at address \(X\) in the binary.

In complex (CISC) instruction sets some instructions may contain implicit control flow. For example, the x86 `test` instruction sets the value of the `eflags` register based on the value of its operands. The LLVM IR is a reduced (RISC) instruction set which needs an explicit branch instruction to implement the same behaviour, resulting in multiple LLVM basic blocks to emulate a single x86 basic block. To capture these basic blocks into a single unit of execution, \(S^2E\) uses a single LLVM function to represent a single basic block of the target architecture. Section 5.1.1 expands on the translation model of \(S^2E\) with an example. We have developed a plugin that copies generated functions into a new LLVM module. This module is the input for the code transformations described in Chapter 5.

### 4.3.1 Enforcing creation of LLVM IR

Recall from Section 4.1 that \(S^2E\) uses selective symbolic execution which switches between DBT mode and symbolic mode depending on the presence of symbolic values in memory. Only when running in symbolic mode will LLVM code be generated, in order to be passed to the KLEE executor. In DBT mode, the instrumented QEMU IR is translated to machine code for native execution. This poses a problem for our framework which attempts to export LLVM for all executed code of a program, also before execution has been forked and symbolic mode is engaged. Furthermore, the framework should be able to export code when symbolic execution is limited or disabled altogether, for applications that only require a subset of the code to be exported.

The \(S^2E\)-adapted version of QEMU provides the `--generate-llvm` configuration option which causes LLVM to be generated for all executed code, regardless of the execution mode. This causes a decrease in performance of an order of magnitude. This overhead is unnecessary because a large part of the executed code is library code or code running in the background, for which the LLVM is not exported. We therefore inspected the internals of \(S^2E\) to find the code responsible for generating LLVM IR. The framework uses this code to only generate LLVM on demand when it needs to exported

\(^2\)A basic block is a sequence of instructions that is always executed contiguously, with a single entry point and a single exit point.
and does not use the `--generate-llvm` option. Logically, this optimization increases performance back an order of magnitude, allowing for analysis of bigger programs.

### 4.3.2 Recording basic block successors

The S2E executor decides which basic block to execute based on the value of the program counter register, which is a member of the virtual CPU state. Control flow in the generated LLVM is implemented by setting the starting address of a basic block in this register. This address is peculiar to the address space of the original code. When lifted to LLVM IR, however, these addresses lose their meaning since basic blocks in LLVM do not have numeric addresses. Control flow instructions instead require a direct pointer to a targeted basic block. In order to reconstruct control flow, a mapping is needed between program counters and basic blocks.

For direct jumps and branches, this mapping can be recovered by matching the saved program counter with the label of an exported basic block, which contains its address observed at runtime. For indirect jumps, e.g. `jmp eax`, the saved program counter is not constant and the corresponding basic block to jump to cannot be determined statically. Creating a branch to all possible successor basic blocks requires a mapping of all possible program counters at that point to corresponding basic blocks. When assuming that the program counter may take an arbitrary value, the mapping must contain all possible program counters in the program, which is a vast over-approximation. To be able to limit the complexity of the code, our plugin records a list of successors for each basic block. It does so by remembering the previously executed basic block for each program state, and adding the currently executed block to its list of successors in the form of an LLVM metadata annotation. Section 5.1.4 explains how these annotations are converted into explicit control flow instructions.

The set of successors observed at runtime is an under-approximation unless all possible code paths are covered by symbolic execution. This means that for the cases in which less than 100% code coverage is achieved, a successor list may be incomplete. Such a case may occur when symbolic execution is terminated before finishing all code paths, or when symbolic execution is disabled to capture a single-path variant of the binary. For code that is never executed during symbolic execution, this is not an issue. If a successor list is complete and if the code corresponding to it was not fully explored, the code path cannot be recovered after all. In cases where one of the possible targets of an indirect jump is already executed in another code path and therefore already exported, but not observed as a successor of the current basic block, over-approximating the
successor list by including all exported basic blocks might lead to better accuracy. However, we have chosen not to over-approximate because the benefit of having small successor lists outweighs the chance of having slightly better accuracy for edge cases.

Note that by mapping each program counter to a single basic block, we assume that the mapping is unique for the entire duration of a program’s execution. It is easy to conceive a program in which this is not the case. Consider the example of a packer from Section 4.2.2 that dynamically extracts code. The packer allocates a buffer on the heap, unpacks a basic block into that buffer, executes it and unpacks another basic block into the buffer for execution. In this example, the address of the buffer can map to two different basic blocks, invalidating the assumption. A possible solution to this problem could record to a mapping for each basic block, rather than a single global mapping (see also Section 8.1). Our framework currently maintains a global successor list, hence it does not support this type of cases.

4.4 Exercising the binary

Symbolic execution only forks execution when a symbolic value decides the which direction of a branch to follow. Therefore, it is important to symbolize values that influence control flow. In program testing applications, when there is access to source code, instrumentation functions may be called in the code to make specific variables symbolic and thus specifically target a certain piece of code to discover. When analysing a binary of which the source is inaccessible, little or no assumptions can be made about its internal memory structure. This limits the choice of values to symbolize to the external environment from which the binary gathers its inputs. For example, command-line arguments, environment variables and opened files. Our framework currently supports symbolic command-line arguments for Linux- and Windows-based binaries (ELF and PE binaries respectively).

For Linux, init_env.so offers arguments similar to KLEE in order to insert symbolic arguments in the form of --sym-args and --concolic, allowing for trivial insertion of symbolic values for command-line arguments. For Windows, no such arguments exist. This is likely due to the difference in the way both operating systems pass command-line arguments to programs; Linux places them on the stack, whereas on Windows a library call to kernel32.dll::__getmainargs is required to retrieve the arguments. We have developed a custom plugin that intercepts calls to this function and inserts concolic values for the returned arguments.
5. TRANSFORMATION OF CAPTURED CODE INTO EXECUTABLE BINARY

This chapter explains in detail which code transformations are performed on the LLVM IR captured by our framework.

In Section 5.1, dependencies on the S²E runtime are removed from the raw LLVM code. Non-essential instrumentation (e.g., helper functions) is pruned. Then the remaining dependencies on the S²E execution state are removed by adding explicit control flow between basic blocks.

In Section 5.2, intermodule dependencies are fixed. Register marshalling code is inserted at library function calls and extern variables are transferred by copying relocations.

In Section 5.3, data dependencies are resolved. Fixed-address pointers from the original code that have been invalidated are made valid again by inserting data sections at the correct addresses.

Finally, Section 5.4 describes finalization of the binary: optimization, bootstrapping and linking. The code has been transformed into a format that can be optimized using the LLVM toolchain without losing semantics. Initialization code is inserted to set up the stack and registers and the binary is linked with flags that exclude libc-generated code and insert data sections at fixed addresses. LLVM arrays are used to improve optimization of memory accesses.

5.1 Removing S²E runtime dependencies

5.1.1 Runtime view of program code

When running in symbolic mode, S²E generates LLVM code at the granularity of basic blocks. The symbolic executor maintains a code cache that maps code addresses to LLVM functions. Each LLVM function in the cache represents a basic block from the input binary. Unlike the basic blocks in the input binary, the functions in the cache are decoupled. That is to say, there is no explicit control flow between functions representing different basic blocks. Instead, the generated code writes a code address
to a global variable, the virtual program counter, which we will refer to as the PC from hereon. The executor reads the value of this variable to determine which function from the code cache to execute next. Figure 5.1 shows an example of a code cache. Note that, as explained in Section 4.3, a branch in the input binary results in multiple LLVM basic blocks, each of which stores a different value to the PC.

Figure 5.1.: An example of a code cache. On the left is a control flow graph showing basic blocks at addresses A to F in the input binary. On the right is the code cache of the corresponding LLVM functions maintained by S\textsuperscript{2}E. The basic blocks at A and B both end with a branch, resulting in multiple basic blocks in the corresponding LLVM functions.

Our S\textsuperscript{2}E plugins export LLVM functions directly from the code cache. These functions adhere to the structure described in Algorithm 1. The “interesting” code that implements the behaviour of the source code comprises instruction bodies and manipulation of the PC for control flow, respectively lines 5 and 9. The event triggers and instruction counter on lines 2, 4, 6 and 8 only exist to provide information to the S\textsuperscript{2}E runtime, and can be removed. The remaining code still contains dependencies on the S\textsuperscript{2}E runtime. Sections 5.1.2 to 5.1.4 describe these dependencies and show how they are removed.

5.1.2 Rewriting accesses to the CPU state

Registers, flags and the program counter are all members of the CPU state, which is passes as the \textit{env} parameter in Algorithm 1. This runtime data structure is not present in the recovered code. The data structure is accessed through the \texttt{getelementptr}
Algorithm 1 Structure of an exported LLVM function. \( X \) is the start address of the exported basic block. \( env \) is \( \text{S}^2\text{E} \)'s internal CPU state. \( pc \) is the offset of the program counter in \( env \), used to implement control flow. The union of all instruction bodies is the main part of the emulated code, this is the part in which we are interested. \( icount \) is the offset of a global instruction counter, which is part of the instrumentation. \( nextpc \) is a constant number if the last basic block ends with a direct jump, otherwise it is a number computed in the instruction bodies.

```
1: procedure Func\_X(env)
2:     emit start event
3:     for each instruction \( i \) do
4:         \( env[pc] \leftarrow \text{address}(i) \)
5:         instruction body
6:         \( env[icount] \leftarrow env[icount] + 1 \)
7:     end for
8:     emit end event
9:     \( env[pc] \leftarrow nextpc \)
10: end procedure
```

instruction of LLVM, which adds an offset to a pointer. Our framework transforms each distinct offset to \( env \) into a global variable. The generated global variables are named using a database of names and corresponding offsets. This database is architecture specific as different CPU state objects are used for different architectures. Code 5.1 shows an example of CPU state rewriting.

Generic variables that are generated regardless of the architecture of the input binary are the program counter (PC), the instruction counter and the current translation block (a container for binary translation). Architecture-specific variables for the x86 architecture are general-purpose registers (e.g., \( eax \)), memory segments (e.g., \( gs \)) and the flags status register \( eflags \).
An example of CPU state rewriting. A pointer to the CPU state at offset 12, corresponding to register `ebx` in the input binary, is transformed into a reference to a newly generated global variable `R_EBX`.

```c
define i64 @Func_8048450(i64 *%env) {
   ...
   ; original code
   %env_ptr = getelementptr i64 *%env, i32 0 ; get pointer to env
   %env_addr = load i64 *%env_ptr ; load env address
   %ebx_offset = add i64 %env_addr, 12 ; add offset 12 for ebx
   %ebx_ptr = inttoptr i64 %ebx_offset to i32* ; cast address to pointer
   %ebx_value = load i32* %ebx_ptr ; dereference pointer
   ...
}
```

5.1.3 Including helper implementations

For complex source instructions that cannot be mapped to corresponding LLVM instructions, S²E provides helper functions that emulate the desired behaviour. An example is the `cpuid` instruction of x86, which reads the `eax` register as an implicit parameter and returns information about the processor in registers `eax`, `ebx`, `edx` and `ecx`. When the code translator encounters this instruction, it inserts a call to a function called `helper_cpuid`. This helper gets processor information from the virtual machine currently running and stores the returned values in the CPU state.

The implementation of this type of helper functions is not in the exported code. Their behaviour is therefore not implemented in the recovered code. This problem is addressed by copying implementations of helper functions from S²E. The `--output-module` configuration option is used to export implementations of all helper functions into a separate LLVM module during execution of the framework. We use this exported module to merge helper implementations into the recovered code. We copy the implementations of two classes of helpers. The first class comprises helpers that compute flags based on instruction opcodes. S²E makes use of the flags helpers whenever flags are explicitly referenced in the binary input code. For example, the `pushf` instruction
pushes the value of the flags to the stack to be read back later using a \texttt{popf} instruction. The second class of copied helpers are those that implement integer division.

Some other helpers cannot be copied blindly, but instead require more transformations. These are helpers that do floating-point operations and (un)locking instructions for multi-threaded applications. In this work, we have chosen not to implement support for these operations due to time and scope constraints.

\texttt{cpuid}  A special case is the \texttt{cpuid} instruction described above. The framework uses a virtual machine in which the binary under analysis is executed. The VM emulates a particular processor which may differ from the processor on the target hardware on which the recovered binary is required to run. Symbolic execution would be able to circumvent this problem by exploring the entire \texttt{helper.cpuid} function and thus covering all different cases, if the returned values would be made symbolic. However, when symbolic execution is disabled in order to recover a single-path variant of a binary, the path taken in the helper in the VM may differ from that taken on native hardware, causing the program to enter an invalid state by jumping to a non-existing basic block (invalid program states will be explained in more detail in Section 5.1.4). To avoid this undesirable behaviour, the implementation of this helper is replaced by one that returns the values corresponding to the processor in our analysis VM image.

\textbf{Exceptions}  Another special cases is created by instructions that raise exceptions, e.g., division by zero. These edge cases are handled by function \texttt{helper.raise.exception}, which propagates the exception to the virtual machine. We have replaced the implementation of this function with a printed notification because such edge cases are not in the scope of this work.

\textbf{Memory references}  A third special case are loads and stores to memory, which show as function calls to \texttt{ldX.mmu} and \texttt{stX.mmu} (where \texttt{X} indicates word size), passing the address to load from or store to as the first argument. These calls are replaced by a load or store of the address from the passed argument with a type cast to match the word size \texttt{X}. However, the LLVM optimization engine is unable to reason about these memory locations because they may be overwritten by any external module. We therefore model program memory in an unbounded byte array called \texttt{@memory}. This is a temporary transformation that will be reverted after optimization. The transformation also allows for more effective analysis, since accesses to memory can be easily identified by looking at loads and stores to offsets of the \texttt{@memory} array.
5.1.4 Fixing control flow

Recall from Section 4.3.2 that a list of successors is recorded for each exported basic block at runtime. This list of successors needs to be converted into an explicit control flow instruction at the end of each basic block. In LLVM, control flow instructions can only target basic blocks within the same function. However, the exported successor lists contain pointers to functions for each exported function. To create explicit control flow instructions, all functions that correspond to basic blocks in the input binary are merged into one large wrapper function. Furthermore, this enables extra optimisations at the function level. For example, basic blocks connected by a single control flow edge can be merged after which aggressive intra-block optimizations can be performed.

After merging the basic blocks from all functions, the exit blocks from the merged functions all end with a return instruction (`ret void`). The entry block of each function is annotated with a successor list, which now contains pointers to basic blocks in the wrapper function, rather than pointers to functions. Each return instruction is replaced with a `switch` statement: an LLVM instruction that implements a branch, choosing a target to jump to based on the value of a conditional expression. We use the target address from the PC variable as the conditional expression. Together with the list of addresses and basic blocks, we can select the target basic blocks of the switch statement.

A switch statement also requires a default basic block to jump to when the expression does not match any of the addresses in a successor list. We insert an error block for this case, which shows that the recovered program has entered an invalid state. In debug mode the inserted error block displays a warning message and in release mode the error block consists just of the `unreachable` instruction. In this way, in release mode, the error block is optimised away by the LLVM engine. This means that only the code paths observed during symbolic execution are supported by the resulting code, and any paths that deviate may lead to undefined behaviour in the optimized release binary.

Code 5.2 demonstrates an example of control flow in which a successor list containing two successors is converted into a `switch` statement. The value of the `@PC` variable is set by the instructions preceding the return, and contains some code address that corresponds to the basic blocks at addresses Y and Z in the input binary. In actual code, X, Y and Z would be numeric addresses, but textual labels are used in the example for better readability. Note that in the optimized code, the value of the `@PC` variable is not loaded anymore. Instead, the condition that previously determined which value was stored to the PC, is now used directly to make the control flow decision,
rather than indirectly through the @PC variable. In our test applications, we see that the @PC variable is removed altogether during optimization and the conditional expressions that determine its value are instead used directly by control flow instructions.

**Code 5.2** An example of fixing control flow. A successor list containing two successors is converted into a switch instruction. Including the error block, the resulting basic block has three possible successors. In release mode the LLVM optimizer will recognize that the error is unreachable, resulting in two possible successors, and inserts a br instruction.

```
; after merging basic blocks from all functions
BB_X: ...
  ret void !(BB_Y, BB_Z)

; switch statements inserted
BB_X: ...
  %pc = load i32* @PC
  switch i32 %pc, label %error_block [
    i32 Y, label %BB_Y       ; %error_block is the default target
    i32 Z, label %BB_Z       ; jump to %BB_Y if %pc equals Y
  ]

; after optimization
BB_X: ...
  %condition = ...
  br i1 %condition, label %BB_Y, label %BB_Z
```

### 5.1.5 Merging overlapping basic blocks

A issue when using the value of the PC to fix control flow, is that of overlapping basic blocks. When the code cache does not contain an entry for a certain program counter, it fetches the machine code from the input binary and creates a basic block containing the instruction at that address, up to and including the next control flow instruction (typically a jump, branch or return instruction). Later the target of a jump turns out to be in the middle of the already created basic block. This address is not yet in the code cache and the executor generates a new basic block containing all instructions up to the next control flow instructions. The code in this basic block
Figure 5.2.: Merging overlapping basic blocks. Basic block $A$ is divided into parts $A'$ and $A''$ where $A''$ and $B$ contain the same instructions. $A''$ is removed, $B$ is set as the only successor of $A'$ and the successors of $A$ (in this case only $C$) are moved to the successor list of $B$.

is entirely contained within the already existing basic block $A$. Such code duplication on itself is not a problem, it would only result in a slightly different control flow graph than in the recovered program. However, when basic block $A$ is executed again in $S^2E$, execution events trigger for both $A$ and $B$, causing $B$ to be listed as a successor of $A$. Since the code in $A$ does not change the PC value to $B$ (but to the successor of $B$), the recovered program will enter an invalid state when $B$ is expected by the generated switch instruction at the end of $A$. To prevent this from happening, overlapping basic blocks are merged by a transformation pass. Figure 5.2 shows how overlapping basic blocks are formed and merged.

5.2 Fixing intermodule dependencies

5.2.1 Calls to extern functions

Calls to library functions do not work in the recovered LLVM code for two reasons. First, the import table from the input binary is not in the recovered code, so dynamically linked function addresses cannot be resolved. Second, registers in the recovered code are converted to global variables, whereas library code expects communication through physical registers.

Reconstructing the import table In dynamically linked binaries, calls to extern functions rely on information from the import table: a list of symbols (or offsets) of
functions created by the linker that is filled in with addresses in the corresponding loaded libraries by the program loader. In ELF binaries this is called the Global Offset Table (GOT). The compiler generates a stub function for each extern function, which loads the address filled in by the linker from the import table and jumps to that address. In ELF binaries, the initial address in the GOT points to an additional stub function in the Procedure Linking Table (PLT) which requests the library address from the loader and overwrites the GOT with the returned value.

The recovered LLVM contains these stub functions which dereference a constant address in the import table. This address is invalidated in the recovered binary which does not share the import table of the input binary. A possible fix is to simply copy the import table from the source binary and link it at the same address. However, the table would have to be extended with any additional functions called by our transformation passes. Moreover, the recovered code would not show which function is called, limiting effective analysis of the code. Instead, our framework reconstructs the function calls at the call sites, building a new import table that contains the same symbols as the input binary.

Function calls are reconstructed in the basic blocks of the recovered stubs. These stubs are identified in different ways for PE (Windows) and ELF (Linux) binaries. PE binaries do not have a PLT, but instead rely on the program loader to fill in the import table when the imported libraries are loaded. Our runtime S²E plugin detects the event that an imported library is loaded and inspects the import table in program memory. When a basic block at an address from the import table is executed, the previously executed basic block is annotated with the corresponding function symbol. For malware analysis, this method is more reliable than static extraction of the symbols, since the import table header is often obfuscated. In ELF binaries however, the PLT fills in the import table (GOT), so extern function addresses are still unknown when the library is loaded. Currently, the framework relies on the GNU objdump utility to extract stub function and their respective symbols statically. If we want high accuracy, a more sophisticated runtime plugin could be used for the detection of linking patterns.

**Marshalling register values at call sites** Function calling conventions use registers to communicate arguments and return values. Registers in the recovered code are converted to global variables in memory, as explained by Section 5.1.2. A library function that is called, however, is not recovered code and as such expects communication through actual registers. Instrumentation code is therefore inserted at function call
sites to copy the values of register variables in LLVM (these will be referred to as virtual registers from hereon) to actual registers, and copy the return value of the function back after it returns. This is best explained by an illustrative example. Code 5.3 shows an example function call. An argument is pushed to the stack at esp and the function *puts* is called. The call instruction pushes the return address (the address of label *return*) onto the stack and jumps to *puts*. Code 5.4 shows the corresponding LLVM function after our helper code is inserted. The inserted code calls a helper function that performs the extern call.

**Code 5.3** Function call *returnvalue = puts(string)*; in x86.

```plaintext
fun call:
    push string
    call puts
return:
    mov [returnvalue], eax
```

**Code 5.4** Function call *returnvalue = puts(string)*; transformed in LLVM. Value types are omitted and temporary variables (starting with %) are named for readability.

```plaintext
fun call:
    %esp = load @R_ESP
    ; Push argument ‘string’ to the virtual stack
    %esp = sub %esp, 4
    store %string, %esp
    ; Push the return address
    %esp = sub %esp, 4
    store 134513759, %esp
    store %esp, @R_ESP

    ; Call the helper function with a pointer to extern function ‘puts’
    ; Use the original return address to decide the next basic block
    %ret = call helperExternCall(@puts)
    store %ret, @PC

    %pc = load @PC
    switch %ret [ ... ]
return:
    %returnvalue = load @R_EAX
```
The helper function is hand-written x86 assembly because it needs to explicitly set values of registers. The helper copies the values of all virtual registers to actual registers before the function call. For most cases it might suffice to only copy 0R_ESP to esp, but it is more reliable to copy all registers. This way, calling conventions that pass arguments in registers for optimization purposes are also supported, and unexpected side effects of library functions are preserved. After the function call, the values of actual registers are copied back into the virtual registers to copy the return value and any side effect operations on registers.

In order to let the library function return to the code that copies registers, the return address on the stack needs to be patched. The return address by default is the address of the instruction following the call in the input binary, which is invalid in the recovered code. This address is saved and later returned by the helper in order to implement control flow as described in Section 5.1.4. Its entry on the stack is replaced with an address at an offset in the helper, namely the code that copies actual register values back to virtual registers. Appendix A shows the x86 assembly source code of the helper function for further reference.

5.2.2 Extern variables

In dynamic object types such as ELF, variables imported from libraries are implemented using dynamic relocations [34]. A dynamic relocation is performed by the program loader, which copies the initialized value of a variable from a library into an uninitialized variable slot of the same size in the .bss section of the target program. All such relocations that need to be performed by the loader are recorded in the relocation table of a binary in relocation entries or “fixups” by the linker. For example, when a program prints a string to its standard error output stream, it imports the variable stderr. The linker generates a fixup of type COPY specifying the symbol “stderr” and the address of a zero-initialized variable in the .bss section. The loader reads the value of stderr from the loaded standard library libc.so and writes it to the address specified in the fixup. When the program wants to write a string to standard error, it reads the variable in its .bss section.

In the LLVM code recovered by our framework, symbols corresponding to imported variables are lost. Corresponding variables in the .bss remain zero-initialized. In order to make imported variables work again, the framework generates a new imported variable for each COPY relocation from the input binary, so that the same relocations will be generated when linking the recovered program. The generated variables are
ordered by their addresses in the relocation table. The first relocation entry is placed at the same address as in the original binary using a linker flag. Subsequent variables are each given the size in bytes of the offset to the previous relocation entry, so that their relative byte offsets are maintained.

In this way, we generate variables that are enforced to be placed at specific addresses. The recovered code only uses these variables indirectly through their addresses. Therefore, the variables appear not to be used by the recovered program code, and they are removed by the compiler during optimization. To prevent this from happening, a dummy function is generated which stores a value to all generated variables.

To read the relocation table from the original binary, the framework currently relies on the GNU `objdump` utility. This is not robust when the file headers are corrupted, which may be the case in malware. In future work, a better approach could be to read relocation entries from memory at runtime, just before the loader reads the relocation table (at which point it must contain valid entries). Alternatively, relocations could even be intercepted in the loader itself.

### 5.3 Fixing data dependencies

Recovered LLVM code accesses data using constant addresses from the address space of the input binary. To make these addresses valid when running the recovered code, the data needs to be present at the same addresses in the recovered binary. This is accomplished using LLVM global arrays and linker flags. A global array is generated for each data section from the original binary. For initialized sections (e.g., `.data`), the array is initialized with a byte array containing raw data read from the binary. The `constant` annotation is added if the section is marked read-only (e.g., for `.rodata`). For uninitialized sections (e.g., `.bss`), the array is marked zero-initialized. Each global array is marked with a unique section name in which it will be contained after compilation. These sections are linked at the same addresses as the sections from the original binary using linker flags.

Code 5.5 shows an example of copied sections in recovered code. Note that all variables are byte-aligned, this avoids padding bytes being added by the linker in order to meet default (typically 16 byte) alignment of sections after compilation. Also note that `.bss` is not renamed in order to support the recovery of extern variables in this section.
An example of copied data sections in recovered LLVM.

@__rodata = constant [22 x i8] c"....", section "__rodata", align 1
@__data = global [8 x i8] zeroinitializer, section "__data", align 1
@.bss = global [4 x i8] zeroinitializer, section ".bss", align 1

To improve optimization and code readability, any constant address reference that maps to a copied readonly section is replaced with a pointer into one of the generated global arrays by means of a `getelementptr` instruction. In the example from Code 5.5 this means that accesses to addresses in the range of `@rodata` are replaced with offsets into that array. This allows for slightly more constant propagation, since values at dereferenced addresses are encoded in the global LLVM array.

5.4 Finalization

5.4.1 Bootstrapping

x86 programs use a stack to store variables in memory. The stack is accessed through the stack pointer register `esp`. In recovered LLVM code, the stack pointer is a global variable called `@R_ESP` as explained in Section 5.1.2. This variable needs to be initialized to a valid stack address in order for stack operations to work. This “virtual” stack must not interfere with the regular stack at `esp`, which is also used in the recovered binary.

A recovered binary contains a newly generated entry point which allocates a buffer on the heap and initializes the virtual stack at `@R_ESP` to the allocated buffer. The current implementation always allocates 16MB, which is well over the needs of most applications. After the stack is initialized, the wrapped function that contains all recovered basic blocks is called. Afterwards, the stack is freed and the program exits gracefully.

In the case of ELF binaries, a new `main` function is generated as well, and any basic blocks in the wrapper that map to compiler-generated setup code are removed. The generated main function emulates a function call to the original main function by pushing command-line arguments (argc and argv) to the virtual stack before calling the wrapper function. It also exits with the status code in `@R_EAX`, the value returned by the old main function.
5.4.2 Optimization

The raw recovered LLVM code is procedurally generated and inefficient, even after removing instrumentation. Much of the inefficiency is due to explicit control flow between basic blocks that is generated when all basic blocks are merged into a single function and successor lists are converted into switch instructions. In many cases, these switch instructions can be turned into direct jumps (when there is one successor) or branches (when there are two successors). Basic blocks connected by a direct jump can subsequently be merged if the successor block is not the target of any other jumps, after which aggressive intra-block optimizations can be performed.

We use the existing LLVM toolchain to perform these optimizations in an effort to reduce code size and improve runtime performance, running opt -O3 -globalopt on the recovered code. The optimization engine is able to remove the program counter variable entirely in most cases. The optimized code instead directly uses the conditional expressions that decide the value of the PC, in order to make control flow decisions.

Optimization on memory values The LLVM optimization engine keeps track of values stored to arrays in memory, and is able to carry out effective constant propagation on these values. When a program is compiled directly from source to LLVM code, data addresses show as variables or offsets in data arrays, enabling optimizations. The recovered LLVM code, however, references constant addresses, about which the optimization can not make any assumptions, and thus can not optimize. In order to make effective optimizations available to recovered code, memory accesses are rewritten to offsets in a global memory array (see also Section 5.1.3) before optimization. After optimization, these offsets are rewritten back to constant addresses.

5.4.3 Compilation and linking

The last step of the recovery process is conversion of the recovered code into an executable binary. Compilation of LLVM code into object file is straight-forward using the llc utility from LLVM’s toolchain. The object file is statically linked with an object file of helper functions that are written in x86 assembly, described in sections 5.1.3 and 5.2.1. The linker receives --section-start arguments with for all copied sections as described in Section 5.3. Dynamically linked libraries are currently specified manually after inspecting the input binary.
6. EVALUATION

6.1 Metrics and test setup

In this chapter, we evaluate four aspects of our framework:

1. Usability: We show how an example binary is handled by the framework using only two commands, along with how its code is transformed from input binary to IR and from IR to recovered binary. The example demonstrates the effectiveness of the framework in a way that is easy to understand and illustrates the interplay of existing LLVM optimizations with our deinstrumentation passes.

2. Accuracy of the recovered binary, verifying that the performed code transformations accurately preserve program semantics. This is measured qualitatively by comparing output of input binaries to their recovered variants.

3. Effectiveness of multi-path exploration. Binaries are recovered with symbolic execution enabled and the code in the resulting binaries compared to the code in the original binaries. This is measured quantitatively by calculating code coverage.

4. Runtime performance of recovered code. We run a number of benchmark programs and compare the execution time of the input binary to that of the recovered binaries, quantifying the result in terms of speedup.

All experiments have been conducted on a machine with a quad-core Intel Xeon CPU E3-1246 v3 running at 3.50GHz. The machine has 32GB memory of which 25GB is available for experiments.

6.2 An example

This section demonstrates ease of use of the framework by showing how an example program passes through the various framework components; from source code, all the way to recovered binary. We will also illustrate the effectiveness of optimizations.
Code 6.1 Source code of example ab program.

```c
// ab.c
#include <stdio.h>
int main(int argc, char **argv) {
    char a = argv[1][0];
    char b = argv[1][1];
    if (a == 'a') {
        if (b == 'b') {
            puts("You entered \"ab\" ");
        }
    }
    return 0;
}
```

Code 6.1 shows the source code of our example program, which is called ab. The ab program checks if the user entered the string “ab” as the first command-line argument. The program contains two branches: the two if-statements that check if the first two characters of argv[1] are ‘a’ and ‘b’ respectively. When argv[1][0] and argv[1][1] (or a and b) are symbolic values, the executor forks the execution at lines 6 and 7 and follows each direction of the branch, resulting in all three code paths being discovered.

Figure 6.1 illustrates the code paths explored by symbolic execution of the example program.

Capturing code Symbolic execution is initiated by issuing a single console command:

```
$ qemu/cmd-debian.sh ./ab --sym-arg 2
```

This command creates a configuration file with program name and arguments, loads a preconfigured VM snapshot, copies the configuration file and input binary ./ab into the VM and starts the binary with the init_env.so library preloaded. The --sym-arg 2 argument is interpreted by init_env.so, which replaces the arguments with two symbolic bytes. The command runs until symbolic execution finishes or the VM process is killed. The VM process can be killed either manually by interaction or automatically after a preconfigured timeout or when resources are exhausted. In these cases, the framework will exit gracefully and still produce output of the code paths covered up to that point. An output directory is produced, which contains the raw LLVM code generated by S2E for execution by the KLEE symbolic executor.
Figure 6.1.: CFG of `ab` program with code paths followed by symbolic execution. Edge labels are constraints added to the current execution state after each branch. A state fork occurs when a node has two successors.

Figure 6.2.: Raw code exported by S2E plugins. Input basic blocks are not connected by explicit control flow edges. The edges that are present implement control flow of branching x86 instructions.

**Transforming captured code into recovered binary** Another command is needed to transform the code captured by S2E into a working binary:

```
$ recover/recover.sh s2e-out-ab/ ab-recovered
```

This command runs all LLVM transformation passes described in Chapter 5 and compiles and links the code in output directory “s2e-out-ab” into a working binary called “ab-recovered”.

**Removing instrumentation** Figure 6.2 shows a CFG of the captured code, in which explicit control flow between basic blocks is replaced with implicit control flow through the virtual program counter (as explained in Section 5.1.1). Figure 6.3 shows the code...
of a single input basic block (which maps to multiple LLVM basic blocks, each exit block storing a different program counter), corresponding to two x86 instructions. It is easy to see that the code is heavily instrumented by S²E, using 65 LLVM instructions to emulate two x86 instructions. After removing instrumentation as described by Section 5.1, the code is reduced to Figure 6.4(a). Note that explicit control flow instructions have been added to connect basic blocks. After built-in LLVM optimizations are applied, only a single LLVM basic block remains as shown in Figure 6.4(b) containing 4 instructions, 16 times fewer than the raw code.

\begin{verbatim}
    cmp eax, 1
    jle label
\end{verbatim}

Figure 6.3.: Raw captured LLVM code corresponding to two x86 instructions. The LLVM code on the right is not intended to be readable, it is displayed only to demonstrate the size of captured code.

**More optimization** Because basic blocks that implement control flow are optimized away as shown by Figure 6.4(b), individually optimized basic blocks that are connected can be merged as well. Figure 6.4 shows how intra-block optimizations cascade to merging of basic blocks. This again allows for aggressive intra-block optimization in merged basic blocks, further reducing code size. We did not need not implement these any of these optimizations ourselves, since they were readily available in the LLVM toolchain.
(a) Basic block from Figure 6.3 after instrumentation is removed as per Section 5.1.

(b) More code reduction after built-in LLVM optimizations.

Figure 6.4.: Cascading optimizations by the LLVM optimizer after optimizing individual basic blocks.

**Resulting code size** After transformations and optimizations, the size of the recovered code is greatly reduced. Figure 6.5 compares the recovered ab program to the output of the clang [35] compiler. The shape of the CFG is exactly the same, indicating a successful recovery of control flow. The recovered code still shows the effects
of instrumentation; it contains visibly more instructions than the code constructed by
clang. Where clang uses temporary variables, the recovered code is stuck with registers
from the original program stored in global variables. Chapter 8 makes suggestions for
improvements in this area.

Summary Using only two console commands, a newly crafted binary has been exe-
cuted in a VM with configurable symbolic arguments, and transformed into a working
recovered binary by a number of LLVM passes.

6.3 Single-path recovery

Table 6.1 shows a list of binaries for which our framework successfully recovered single-
path variants. These binaries have been selected to test the various code transforma-
tions described in Chapter 5. Some of these binaries contain control flow that depends
on input values. Single-path execution yields an incomplete view of a binary that supports only the code path seen during that particular execution. We have manually selected sets of program arguments for these programs such that all or most code is covered by the union of all execution, so that accuracy is tested for at least a representative set of the different code paths.

`ab`, `fibonacci` and `pwcheck` are simple toy programs that contain small test cases used during development.

`pwcheck_vm` is a custom virtualization based obfuscated binary which contains a small bytecode interpreter written in C and corresponding bytecode as data. The complex control flow of the of the interpreter loop is successfully reconstructed. `ab_codevirt` is another virtualization-obfuscated binary, obfuscated using CodeVirtualizer [9], a state-of-the-art industry grade binary obfuscator. We used the evaluation version for Windows (PE) binaries which corrupts the import table and increases the file size by a factor 6. Nonetheless, our framework was able to successfully recover the binary, reconstructing indirect control flow edges using successor lists.

`sha512` and `dhrystone` are used for runtime performance benchmarks in section Section 6.5. They perform a series of calculations to stress the processor. They contain various integer calculations such as bitwise operations on partial values (half-words) which are successfully recovered.

The remaining binaries are command-line utilities from GNU Coreutils [36]. These are real-world binaries that extensively use library functions (and variables). Their successful recovery shows that the framework covers practical edge cases that occur in such binaries, and that the framework is able to recover complex real-world binaries in practice.

### 6.4 Multi-path recovery

Table 6.2 shows a list of binaries we executed with symbolic command-line arguments. Code coverage is measured in two ways:

1. Block coverage: the percentage of all basic blocks in code sections in the original binary for which recovered code is present in the recovered program. This is a pessimistic estimate of the percentage of covered program behaviour, since a portion of the basic blocks may be compiler-generated code or statically linked library code. A number of compiler-generated functions are not counted in the
Table 6.1: Successfully recovered single-path programs. Descriptions include program features for which the binary is selected. ELF binaries are compiled with gcc v4.8.4 and PE binaries with MinGW’s i586-mingw32msvc-gcc v4.2.1.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>ELF</th>
<th>PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>Checks if the user entered “ab” in argv[1]. tests: command-line arguments, simple control flow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fibonacci</td>
<td>Calculates the Nth Fibonacci number. tests: simple control flow with loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwcheck</td>
<td>Checks if the input password matches a constant string. tests: simple control flow, library functions, data section</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>pwcheck.vm</td>
<td>Manually virtualization-obfuscated version of pwcheck. tests: complex control flow</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ab_codevirt</td>
<td>ab program obfuscated with CodeVirtualizer. tests: complex control flow, header obfuscation</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>sha512</td>
<td>Computes SHA-512 hash of input string. tests: complex integer operations, performance</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>dhrystone</td>
<td>Dhrystone [37] benchmark suite. tests: performance</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>env</td>
<td>Utility programs from Coreutils 8.23. tests: complex control flow, library functions, dynamic relocations, data dependencies</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

calculation: code replaced by init_{env}.so (see Section 4.4) and stubs generated for extern functions.

2. Amortized coverage: the percentage of basic blocks from functions whose implementation is in the source code file. This gives a more accurate indication of how much of the program behaviour is covered, i.e., the subset of program behaviours encoded in the source code that are supported by the recovered program. For example, for Coreutils echo we measure the coverage of functions that are defined in the source file echo.c. Library or utility functions that are implemented in other source files or header files are not incorporated in the amortized coverage.

For the toy binaries ab and pwcheck that exhibit simple control flow with regards to input values, full amortized coverage is obtained within a very short time frame because
only a small amount of code paths need to be covered. The 93% and 84% block coverages are caused by compiler-generated setup and teardown functions.

Interestingly, the number of code paths of the virtualization-obfuscated variants `ab_codevirt` and `pwcheck_vm` does not increase. This indicates that symbolic execution might be suitable for deobfuscation of such binaries. Chapter 8 expands on this hypothesis. The block coverage does decrease because the interpreters contain error-checking code for edge cases that are not explored when only symbolizing command-line arguments. No amortized coverage can be computed for `ab_codevirt` because no source code is available.

For the more complex `echo`, the analysis used up all 25GB of available memory after four hours, and execution was halted. `echo` has only 14% block coverage, which can be explained by the fact that many functions in `echo` are statically linked functions common to all coreutils binaries. For example, functions that escape quoted arguments for printing command-line arguments, and generic memory allocation functions such as `xmalloc`. The amortized coverage instead only considers functions that are implemented in `echo.c` in the Coreutils codebase, and is significantly higher at 60%.

Table 6.2: Code coverage and framework runtime with symbolic execution enabled. `--sym-args x y z` inserts `x` to `y` symbolic arguments of size `z`. `--concolic` makes all the following arguments concolic (i.e., it leaves the concrete values but also records symbolic shadow expressions).

<table>
<thead>
<tr>
<th>Program</th>
<th>Arguments</th>
<th>Nr. of paths</th>
<th>Block coverage</th>
<th>Amortized coverage</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ab</code></td>
<td><code>--sym-args 0 1 2</code></td>
<td>5</td>
<td>93%</td>
<td>100%</td>
<td>6 s</td>
</tr>
<tr>
<td><code>ab_codevirt</code></td>
<td><code>--concolic xy</code></td>
<td>3</td>
<td>50%</td>
<td>-</td>
<td>8 s</td>
</tr>
<tr>
<td><code>pwcheck</code></td>
<td><code>--sym-args 0 1 10</code></td>
<td>26</td>
<td>84%</td>
<td>100%</td>
<td>12 s</td>
</tr>
<tr>
<td><code>pwcheck_vm</code></td>
<td><code>--sym-args 0 1 10</code></td>
<td>26</td>
<td>78%</td>
<td>94%</td>
<td>14 s</td>
</tr>
<tr>
<td><code>echo</code></td>
<td><code>--sym-args 0 1 4 --sym-args 0 2 2</code></td>
<td>14567</td>
<td>14%</td>
<td>60%</td>
<td>3 h</td>
</tr>
</tbody>
</table>

6.5 Runtime performance

We have compared the runtime performance of benchmark programs to that of their recovered variants. Table 6.3 shows the results. The normalized runtime in the fourth column is the runtime of the recovered binary relative to that of the input binary. Since recovered code converts registers to memory locations, one would expect recovered
binaries to have significantly degraded performance. This is, however, not the case for our benchmark binaries.

The Dhrystone [37] v2.1 benchmark program performs a number of calculations called “dhrystones” and measures the average number of dhrystones computed per second after a given amount of time. We have recovered an optimized and an unoptimized version of the program, dry2 and dry2_noopt respectively. The recovered binary of the optimized version suffers a 76% performance loss with regards to the input binary. Although this slowdown is significant, it is well within what can be expected from once heavily instrumented code. For the unoptimized version, the recovered binary only loses 20% performance. This can be explained by the fact that the LLVM optimization engine was able to do optimizations on the recovered binary similar to the source-level optimizations performed by gcc on the optimized dry2 binary. Because dry2_noopt does not contain these optimizations but its recovered version does, the performance loss is limited.

The sha512 program iteratively computes a SHA-512 hash of a buffer initialized with a command-line argument, overwriting the buffer with the hash digest after each iteration. The recovered binary is even faster than the input binary in this case, by 15%. The LLVM optimization engine was able to optimize the abstract IR better than gcc was able to optimize the assembly generated directly from source code. This is not only a demonstration of the power of the LLVM optimization toolchain, but it also indicates that even high-performance applications may benefit from recompilation after binary recovery, making binary optimization a possibly interesting area for future research.

Table 6.3: Runtime performance of recovered programs. dry2 and sha512 are compiled with -O3, the highest optimization level. dry2_noopt is compiled without optimizations using -O0. All programs are compiled using gcc 4.8.4. ns/d are nanoseconds per Dhrystone (lower is better).

<table>
<thead>
<tr>
<th>Program</th>
<th>Original</th>
<th>Recovered</th>
<th>Normalized runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>dry2 (200M passes)</td>
<td>33 ns/d</td>
<td>58 ns/d</td>
<td>1.76</td>
</tr>
<tr>
<td>dry2_noopt (200M passes)</td>
<td>75 ns/d</td>
<td>90 ns/d</td>
<td>1.20</td>
</tr>
<tr>
<td>sha512 (5M hashes)</td>
<td>9.7 s</td>
<td>8.4 s</td>
<td>0.87</td>
</tr>
</tbody>
</table>
7. RELATED WORK

Dynamic binary rewriting for analysis purposes is implemented by PIN [38] and Dyninst
[39]. These tools offer an API for dynamic instrumentation, using a custom IR for low-
level code rewriting. Existing static binary rewriting frameworks, including UBQT
[40] and Diablo [41], have also developed their own IR to transform code according to
their different purposes. Our framework instead uses LLVM, allowing users to plug in
existing compilation passes without modification. Furthermore, we separate capturing
code and instrumentation into individual steps. Code capturing occurs dynamically,
with using the advantage of a runtime program view that is not available during static
disassembly. The subsequent transformations are performed off-line. This provides
more flexibility since the user is not constrained to a particular runtime environment,
and allows for time-consuming transformations that would otherwise hinder runtime
performance of the analysis.

LLBT [42, 43] is a static binary rewriting framework based on LLVM, designed for
retargeting a binary to another ISA. Its architecture is similar to our framework, with
the main difference being a static front-end disassembler where we use S²E and dein-
strumentation passes to capture LLVM code.

Another work close to this thesis is SecondWrite [6], also an LLVM based binary rewrit-
ing system. SecondWrite relies on static decompilation of the input binary to obtain
an initial assembly representation of the program. The assembly is then converted to
LLVM by a translator whose output is similar to the code recovered by our framework.
The paper in particular discusses the stack pointer ESP which is, like in our recovered
code, converted to a global variable. From the global stack, individual function stack
frames and function parameters are reconstructed. Value Set analysis [44] is used to re-
liably identify variables in function stack frames, promoting stack offsets to individual
symbols. Our framework does not implement stack frame reconstruction and symbol
promotion, but distinguishes itself from SecondWrite by dynamically capturing source
code. This increases the set of target binaries, in particular obfuscated malware that
violates assumptions made by static decompilation. For example, indirect control flow
instructions are assumed to always have absolute address operands, an assumption
that can easily be violated by obfuscation techniques. SecondWrite was not tested
against binaries with obfuscated control flow, whereas our framework has successfully recovered virtualization-obfuscated code.
8. DISCUSSION AND FUTURE WORK

8.1 Improvement of framework features

Performance of symbolic execution  This thesis focuses on lifting executable code to LLVM and lowering LLVM code back into an executable binary. We have integrated symbolic execution within our framework, but have not explored possibilities for optimization techniques. Future research in this area may include the application of path-finding heuristics to increase code coverage, optimization of memory usage of S²E internals to allow for more state forks and replacement of the standard C library with uClibc [45] in the analysis VM to avoid unnecessary state forks.

Code locations called by library functions  `atexit()` registers a function pointer to be called by `exit()` when the program exits gracefully. In recovered code, this function pointer is invalidated because code locations are changed. This could be solved by inserting jumps at function addresses in the original `.text` section, with the target address being the recovered version of the function in the recovered code section. This effectively transforms the `.text` section into a jump table of all original program functions.

Eliminating fixed address data dependencies  Absolute addresses of data from data sections could be transformed into array offsets in recovered LLVM, using the global variables created for included sections (see Section 5.3). Conversion of absolute addresses opens up possibilities for advanced analysis such as data structure detection.

Self-modifying code  Self-modifying code may replace the code at some program counter with new code, invalidating the one-to-one mapping of program counter to basic block assumed by our framework (see Section 4.3.2). A solution to this problem would need to record to a mapping for each basic block, rather than a single global mapping.

Support for floating point and multi-threading  S²E generates complex instrumentation to support floating point operations and locking operations for multi-threading. Section 5.1.3 describes how and why we do not support these operations. A future version of our framework could potentially convert the instrumentation helpers to LLVM instructions in order to support a wider range of target programs.
Support for more architectures  The framework currently only supports x86 binaries. The target set of binaries could be greatly extended by implementing support for more architectures, for example ARM and x86_64.

ASLR  We have not experimented with Address Space Layout Randomization. We suspect that either absolute addresses would show as offset calculations rather than constant addresses in recovered code of simple ASLR-enabled binaries, or that a randomized offset is encoded into LLVM at runtime. In both cases, detecting and exporting the address space offset in the S²E-generated code or in the S²E runtime would likely solve the problem, since a code transformation pass could use this offset to remove the effect of ASLR from the recovered code.

8.2 Integration of existing binary rewriting techniques

Stack frame reconstruction and symbol promotion  The stack frame reconstruction and symbol promotion algorithms employed by SecondWrite (see Chapter 7) could be applied to the code recovered by our framework. Deconstruction of stack frames would resolve function arguments at library calls, eliminating the necessity of some of the register marshalling described in Section 5.2.1.

ISA retargeting  In Chapter 7 we discussed LLBT, an LLVM based ISA retargeting approach that statically rewrites an ARM input binary to a target ISA. We believe our framework can be a first step towards dynamic ISA retargeting using a back-end similar to that of LLBT on recovered LLVM code.

Binary optimization  Another possible application of our framework is binary optimization. For example, consider a script that uses an awk script to process a large data sets, requiring high performance. Our framework could capture the single execution path of awk followed when executing the input script, and the captured code compiled to an optimized binary that only supports that particular input script. To make the resulting binary still work with its regular inputs, an additional LLVM pass could be constructed that makes the recovered code fall back to code from the original binary when a followed code path is not in the recovered code, effectively optimizing a single code path in the binary.

8.3 Control flow deobfuscation

The original motivation for the creation of our framework was a proposal to use symbolic execution as a method of deobfuscation for virtualization-obfuscated binaries. We
first conjectured this idea in a literature study on virtualization-deobfuscation [46]. Existing deobfuscation approaches [47, 48, 49] capture a single code path in an execution trace and deobfuscate that trace by separating emulator control flow from bytecode control flow, using bytecode control flow to construct a deobfuscated subtrace or CFG. The code produced by these methods is not executable and only covers a subset of possible behaviours of the input malware. We suspect symbolic execution to be suited for deobfuscation because it not only covers multiple program paths, but also because branches on input values can be easily identified using state forks on symbolic input. Exploratory experiments [50] have shown promising results.
9. CONCLUSION

In this thesis, we have described a framework for multi-path based binary recovery. The framework lifts executable machine code to LLVM IR, a compiler-level IR that is widely supported by analysis and transformation tools. After potentially applying such transformations, the lifted IR is lowered back to an executable binary. Integration of selective symbolic execution enables multi-path exploration of binary code. To our knowledge, previous research has not attempted to combine symbolic execution with code lifting to LLVM IR.

We have based our implementation on the S²E framework, which lifts binary code to LLVM IR. The lifted code is highly instrumented and stripped of explicit control flow. We have developed deinstrumentation passes that successfully revert these operations. Lifted code is lowered back to a binary executable, which requires two types of dependencies to be preserved. First, location dependencies of data, which are introduced at compile time. Second, dependencies on libraries, which are introduced at link time. Chapter 5 shows how these dependencies are correctly preserved.

Only two basic commands are needed to recover a binary. The first command initiates symbolic execution of the input binary and captures any executed code as LLVM IR. The second command converts the raw, instrumented LLVM IR to clean, connected and optimized IR and compiles the IR into an executable binary.

We have evaluated the accuracy of our framework using a number of input binaries, including several real-world binaries from GNU Coreutils and a binary obfuscated by a state-of-the-art obfuscator. These binaries were successfully recovered, yielding executable binaries that exhibit the same behaviour for all tested inputs. Symbolic execution yielded complete or near-complete code coverage on four toy programs, including two virtualization-obfuscated programs. We obtained 60% source code coverage of Coreutils echo. We recognize opportunities for improvement of code coverage and efficiency of symbolic execution in our framework. Such improvements are not in the scope of this thesis, which focuses on binary recovery, but previous research on symbolic execution has yielded heuristic-based techniques that can likely be applied to our framework.
We have also measured the execution time of recovered code. We obtained an acceptable 78% slowdown on the Dhrystone benchmark, and a 15% speedup on a SHA-512 hashing program. These results are encouraging to explore the possible application of our approach for binary optimization purposes. For example, by optimizing a single code path of a binary that is often used, falling back to the original code for other code paths.

Finally, the original motivation for the creation of our framework was the conjecture that symbolic execution serves as a suitable method of deobfuscation for malware whose control flow has been obfuscated. With this framework, we have provided a foundation for future research in this area.
APPENDIX
A. X86 FUNCTION CALL HELPER

; funcall-helper.asm
; Intel assembly syntax is used in this file
bits 32
cpu 386

extern R_EAX
extern R_EDX
extern R_EBX
extern R ECX
extern R_ESI
extern R EDI
extern R ESP
extern R EBP

section .bss
retaddr: resd 1
target: resd 1
saved.esp: resd 1

section .text

; i32 @helper_extern_call(i32)
global helper_extern_call
helper_extern_call:
 ; argument at [esp + 4] contains address of extern function
mov eax, [esp + 4]
mov [target], eax

 ; backup regs (eax is now tainted but it is used for the return value anyway)
pushad
mov [saved.esp], esp

 ; the virtual return address should be on top of the virtual stack due to
 ; the emulated function call
mov eax, [R_ESP]
mov eax, [eax]
mov [retaddr], eax
; set virtual register values in concrete registers
mov eax, [R_EAX]
mov edx, [R_EDX]
mov ebx, [R_EBX]
mov ecx, [R ECX]
mov esi, [R_ESI]
mov edi, [R_EDI]
mov esp, [R_ESP]
mov ebp, [R_EBP]

; call to extern function: push retaddr; jmp target
; the new return address replaces the one on at [R_ESP], which has been
; backed up in [retaddr]
mov dword[esp], .after_jump
jmp [target]

.after_jump:
; carry concrete register changes back to virtual registers
mov [R_EAX], eax
mov [R_EDX], edx
mov [R_EBX], ebx
mov [R ECX], ecx
mov [R_ESI], esi
mov [R_EDI], edi
mov [R_ESP], esp
mov [R_EBP], ebp

; restore regs
mov esp, [saved esp]
popad

; return virtual return address to set in @PC
mov eax, [retaddr]
ret
LIST OF REFERENCES


[34] TIS Committee et al. Tool interface specification (tis) executable and linking format (elf) specification, 1995.


