UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Dynamic Program Analysis Enhanced Binary Recompilation

Permalink
https://escholarship.org/uc/item/6vp859hn

Author
Parzefall, Fabian

Publication Date
2024

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Dynamic Program Analysis Enhanced Binary Recompilation

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Fabian Parzefall

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ardalan Amiri Sani
Professor Brian Demsky

2024

© 2024 Fabian Parzefall

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

VITA **viii**

ABSTRACT OF THE DISSERTATION **ix**

1 Introduction **1**

2 Binary Recompilation **5**
2.1 IR-Level Program Recovery . . . 6
2.1.1 Disassembly and Control-Flow Recovery . . . 8
2.1.2 Function Boundary Detection . . . 14
2.1.3 Variable Identification and Symbolization . . . 18
2.2 Preservation of Program Semantics . . . 22
2.2.1 Instruction Emulation . . . 23
2.2.2 CPU Switching . . . 26
2.2.3 Load-/Store-Ordering . . . 28

3 Wytiwyg—Dynamic Stack-Layout Recovery for Binary Recompilation **31**
3.1 The Emulated Stack Analysis Hazard . . . 32
3.2 Usage-Based Variable Symbolization . . . 35
3.3 Overview of Wytiwyg . . . 38
3.4 Dynamic Stack Symbolization . . . 40
3.4.1 Stack Reference Identification . . . 40
3.4.2 Object Bounds Recovery . . . 43
3.5 Implementation . . . 50
3.5.1 Function Recovery . . . 50
3.5.2 Variable Argument Library Calls . . . 52
3.5.3 External Functions . . . 53
3.5.4 Non-Deterministic Code . . . 54
3.5.5 Untangling Large Stack Frames . . . 54

3.6 Evaluation 56
3.6.1 Functionality 57
3.6.2 Performance 59
3.6.3 Splitting Accuracy 62

4 Polynima—Practical Hybrid Recompilation for Multithreaded Binaries 64
4.1 Lifting Non-Deterministic Programs 65
4.2 Design and Implementation 66
4.2.1 Compatibility 67
4.2.2 Control Flow Recovery 68
4.2.3 Callbacks 71
4.2.4 Per-Thread Stack 73
4.2.5 Handling Atomic Instructions 73
4.2.6 Non-Atomic Loads and Stores 75
4.3 Evaluation 76
4.3.1 Environment and Software 77
4.3.2 Comparison with Other Lifters 78
4.3.3 Compatibility and Performance 78
4.3.4 Lifting Time 83

5 Applications 87
5.1 Built-in LLVM Transformations 87
5.2 Thread Escape Analysis 89
5.3 Pointer-Identification 91
5.4 Cross-Recompilation 93

6 Discussion 95
6.1 Binary Compatibility 95
6.2 Coverage 97
6.3 Type-Level Rewriting 98

7 Conclusion 100

Bibliography 102

Appendix A Example Programs 107

LIST OF FIGURES

Page

2.1 Overview of the different analysis tasks involved in lifting a binary to a compiler-level intermediate representation. 7
2.2 Possible interpretation of a sequence spanning four bytes within the instruction stream. 9
2.3 Example of an intentionally overlapping instruction sequence. 10
2.4 Example of a program's control flow graph without functions (left) and with functions (right). 15
2.5 Process image of a recompiled binary. 25
2.6 Lifted IR for a call to `memcpy` on 32-bit assembly. 27

3.1 Visualization of the stack frame as allocated by the compiler corresponding to the assembly in Listing 3.2. 35
3.2 Overview of Wytiwyg. The upper section corresponds to the original BinRec recompiler. The lower section outlines our contribution. The bold transitions correspond to the *Refinement Lifting* process. 39
3.3 Overview of our tracing runtime. 45
3.4 Normalized runtime of input (∗) binaries, binaries recompiled and symbolized with Wytiwyg (†), and binaries recompiled and symbolized with SecondWrite (‡) relative to the runtime of the respective binaries compiled and optimized with GCC 12.2. 61
3.5 Accuracy of Wytiwyg. 63

4.1 Overview of Polynima. Dashed lines indicate optional steps. 67
4.2 Lifting times for BinRec's Incremental lifting v/s Polynima's Additive lifting for `401.bzip2` 85

LIST OF TABLES

Page

3.1 Normalized runtime of recompiled binaries relative to the runtime of their respective input binary for each configuration. 58

4.1 Supported Benchmarks. Lasagne builds on top of mctoll. 79
4.2 Performance of Polynima recompiled binaries on the Phoenix benchmark suite. Results in the NA columns report performance if loads and stores are not lifted as atomic instructions. 80
4.3 Performance of Polynima recompiled binaries on the gapbs benchmark suite. . . . 81
4.4 Performance of the original and the recompiled output (in terms of number of clock cycles required) on the latency tests in CKit. 82
4.5 Lifting Times (in s) the for SPECint 2006 binaries against `ref` inputs and the total number of ICFTs (indirect control flows) recorded in the process 84

ACKNOWLEDGMENTS

First and foremost, I extend my sincerest thanks to my advisor, Michael Franz, for his tireless support, guidance, and patience. I am deeply grateful for your insights and expertise, which had a profound impact on my research and my growth as a scholar. By inviting me to this program, you granted me a once-in-a-lifetime opportunity to not only join a research group, but a diverse community of like-minded people. Your commitment to your students' success above everything else is truly admirable. For these, and many other reasons, I am honored and proud to complete my PhD under your mentorship, which has been one of the most rewarding experiences in my life.

I also would like to express my gratitude to Peter Fröhlich. None of this would have ever happened if you had not pushed me to consider joining a doctoral program. I will always be thankful for your confidence in my ability to succeed on this path.

I would also like to acknowledge all my fellow lab mates who I had the pleasure to work with: Joe, Alex, Prabhu, Anil, Taemin, Dokyung, Paul, Min, Mitch, Matt, Dixin, Chinmay, André, Hongyu, Tian, Jeffrey, Billy, Nick, Mahbub, Weitao and James. I especially want to thank Joe, Anil, and Prabhu for building the foundation for my research and providing the initial direction for my contribution. I also would like to express my gratitude for the mentorship I have received from Prabhu and Paul. Mitch, Min, and Matt, I could not have asked for greater friends when starting this program with you. My gratitude also extends to Erika and the rest of Mitch's family for welcoming me into their homes immediately after relocating to a different continent. And Chinmay, thank you for the countless hours helping me to make my research work. It was a great experience working with you on these problems, and I believe you already know how much you contributed to this success. Thank you to all our postdocs Yeoul Na, David Gens, Adrian Dabrowski and Felicitas Hetzelt, for your patience with all my questions. I have learned a significant amount from your feedback and contributions to my research.

Above all others, I would like to thank my wife, Amandeep, for sharing this journey with me. Words cannot express how grateful I am for your unwavering support, especially in all the times I have doubted myself and you have led me back on the path. I could have not done this without you. You are truly the world to me.

Finally, I want to express how grateful I am for my family for giving me the opportunity to take any path in life, for always supporting me and Amandeep unconditionally, for providing a safety net, and for cheering for all our successes.

Thank you.

Portions of Chapter 3 is a reprint of the material as it appears in *What you trace is what you get: Dynamic stack-layout recovery for binary recompilation*, with permission from Fabian Parzefall, Chinmay Deshpande, Felicitas Hetzelt and Michael Franz. Portions of Chapter 4 is a reprint of the material as it appears in *Polynima — Practical hybrid recompilation for multithreaded binaries.*, with permission from Chinmay Deshpande, Fabian Parzefall, Felicitas Hetzelt and Michael Franz.

This material is based upon work partially supported by the Office of Naval Research (ONR) under contracts N00014-22-1-2232 and N00014-21-1-2409, and the Defense Advanced Research Projects Agency (DARPA) under contracts W31P4Q-20-C-0052, N66001-20-C-4027, 140D04-23-C-0063 and 140D04-23-C-0070. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR, DARPA, or any other agency of the U.S. Government.

VITA

Fabian Parzefall

EDUCATION

Doctor of Philosophy in Computer Science **2024**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2020**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Science **2018**
Munich University of Applied Sciences *Munich, Germany*

RESEARCH EXPERIENCE

Graduate Research Assistant **2018–2024**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2019–2021**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Fabian Parzefall, Chinmay Deshpande, Felicitas Hetzelt, and Michael Franz, "What you trace is what you get: Dynamic stack-layout recovery for binary recompilation." In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS '24).

Chinmay Deshpande, **Fabian Parzefall**, Felicitas Hetzelt, and Michael Franz, "Polynima — Practical hybrid recompilation for multithreaded binaries." In *Nineteenth European Conference on Computer Systems* (EuroSys '24).

ABSTRACT OF THE DISSERTATION

Dynamic Program Analysis Enhanced Binary Recompilation

By

Fabian Parzefall

Doctor of Philosophy in Computer Science

University of California, Irvine, 2024

Professor Michael Franz, Chair

Users of proprietary and/or legacy programs without vendor support are denied the significant advances in compiler technologies of the past decades. Adapting these technologies to operate directly on binaries without source code is often infeasible. Replacing existing software stacks is cost-prohibitive and carries the risks of missing functionality, introduction of new bugs, and breaking compatibility with existing deployments.

Binary recompilation is a technique attempting to solve this problem by reusing existing binary executables. By “lifting” existing binary executables to compiler-level intermediate representations (IR) and “lowering” them back down to executable form, this approach enables application of the full range of analyses and transformations available in modern compiler infrastructures. End-to-end binary recompilation is highly desirable, because it promises to bridge the gap between legacy software and modern compiler technologies without requiring access to source code. However, past approaches have not fulfilled this promise. Recompilers lifting binaries using static analysis fail to fully capture the semantics of complex programs and rely on heuristics to recover information that is beyond the reach of their analyses. Dynamic binary recompilers have been able to lift binaries with higher precision. However, they cannot recover local variables in lifted programs, which is a necessary prerequisite for many compiler-related applications, including

performance optimization. Additionally, non-determinism in the execution of lifted programs has been a major obstacle for dynamic binary recompilers.

In this dissertation, we present two novel techniques, Wytiwyg and Polynima, to address the limitations of existing binary recompilers. Wytiwyg employs a dynamic and incremental binary analysis technique to recover function-local variables within lifted binaries. This is accomplished by decomposing the recovery of local variables into a series of instrumentation-based dynamic binary analyses. We show the importance of precise stack variable recovery by recompiling computationally demanding benchmarks of real-world binaries from the SPECint 2006 benchmark suite. Using performance of recompiled binaries as an indicator of IR-quality, our approach significantly outperforms similar recompilers by $1.18x$, on average. Additionally, Wytiwyg accelerates legacy binaries generated by older compilers by an astounding $1.22x$.

Polynima is a hybrid binary recompiler that combines existing static and dynamic binary analysis techniques to recompile multithreaded and non-deterministic binaries reliably while preserving the order of memory accesses as guaranteed by the x86 memory model. By targeting a wide variety of multi-threaded applications, we demonstrate Polynima to be the first recompiler capable of recompiling multithreaded real-world binaries. Our findings show that the high precision of our approach enables Polynima to recompile multithreaded binaries with less overhead than binaries produced by several state-of-the-art recompilers supporting single-threaded binaries only.

Chapter 1

Introduction

Over the past decades, open compiler infrastructures have seen enormous investments by both academic researchers and industry users to advance the analysis, optimization and safety of software. Unfortunately, these advances are often denied to users of legacy binaries. Binaries of programs that can no longer be recompiled from their source code are effectively "stuck in time". This happens when a vendor ceases support of the software, toolchains are unavailable, or the program's source code has been lost. Naturally, not being able to recompile software makes maintenance of legacy binaries very challenging. For instance, users of legacy binaries cannot reoptimize them to utilize features of recent CPUs, easily fix known bugs and vulnerabilities, or deploy sanitizers and mitigations that are readily available in existing compilers. At the same time, replacing legacy software can be very expensive and is often infeasible.

There is a wide body of research in end-to-end static binary rewriting that attempts to address this issue [51]. Many of the existing approaches are quite effective and capable of applying all kinds of program transformations to commercial off-the-shelf (COTS) binaries without requiring user intervention. However, most binary rewriting frameworks have a narrow scope and rarely recover any higher-level program semantics beyond instruction and control-flow information

from the input binaries. Hence, they support only a small subset of program transformations compared to compilers. State-of-the-art rewriters, such as Egalito [52], support alteration of the program's control-flow graph (CFG) and modification of linear instruction sequences, but provide no help in manipulating accesses to variables and their layout in memory.

Binary recompilers attempt to bridge the gap between rewriters and compilers by "lifting" binaries to compiler-level intermediate representations (IRs). Unlike most program representations used by rewriters, compiler-level IRs like LLVM IR [29] encode source-level program structures, such as local and global variables, and types of variables and functions. However, like rewriters, state-of-the-art recompilers usually omit the recovery of most of these structures. This is unsurprising, since the recovery of instructions, control-flow, function boundaries, and global and local variables from binaries are inherently undecidable analysis problems [25]. The primary focus of previous research in binary recompilation has targeted the problem of control-flow recovery and instruction translation from machine code to compiler IRs. Solving these two problems technically enables lifting of binaries to a compiler-level IR and recompiling into a new binary. The usefulness of this approach over plain rewriting is rather limited, as the lifted programs are devoid of IR-level structures that are available when compiling from source code. Many instructions cannot be directly translated to a compiler IR and have to be transformed into sequences of primitive IR instructions to fully capture their semantics. Although the programs can be reoptimized using full compiler optimization pipelines, the lack of IR-level structures makes it difficult to apply many program transformations that rely on this information. This means that the recompiled programs are often larger and slower than the originals without delivering on the promises of program modernization, reoptimization, and hardening.

More importantly, state-of-the-art binary recompilers are unreliable and cannot lift many real-world binaries. Most solutions to lift binaries to compiler IRs usually employ various static program analysis techniques to analyse the target programs [6], [17], [48]. Since the analysis problems required for lifting are undecideable (e.g., to resolve indirect control-flow, or to determine the

size of an object on the stack), heuristics are used to generate the missing information. These heuristics are based on the observation of common patterns in the binary code, need to be tuned for each binary (or even individual functions), and are not guaranteed to yield correct results. The recompiled programs often contain subtle errors that manifest themselves through incorrect behavior on some execution paths, or through unpredictable crashes. To address this shortcoming, research has shifted to approaches that reduce reliance on heuristics by lifting binaries using execution traces [3]. Such approaches restrict the analysis of the program to the observed execution paths. Since this approach removes the need of heuristics for producing precise control-flow graphs, trace-based recompilers can produce lifted programs with higher fidelity. However, the lifted programs still contain no notion of variables. Additionally, limiting analysis to execution traces fails to incorporate paths into the lifted binaries that are not easily executable, such as those that require highly specific inputs, are timing-dependent, or are subject to non-deterministic behaviors at runtime.

This dissertation examines novel approaches in fully automated binary recompilation that address the shortcomings of existing recompilers. Understanding the anatomy of lifted programs as generated by state-of-the-art recompilers is the first step to identify the challenges that need to be addressed to improve the quality of lifted programs. To this effect, Chapter 2 summarizes the various program-analysis tasks in binary recompilation and how imprecisions in these analyses affect the anatomy of the lifted programs. Chapter 3 presents a novel approach, *Wytiwyg*, that uses an iterative process to lift binaries to a compiler IR. Wytiwyg initially uses a trace-based lifter to lift the program to a compiler IR. It then uses a dynamic data-flow framework implemented through a program instrumentation approach to refine the lifted program and recover local variables iteratively. An extensive evaluation demonstrates that this approach can lift binaries with higher fidelity than existing recompilers, and that the lifted programs can be effectively analysed, transformed, and reoptimized with a modern compiler. Chapter 4 presents a novel approach, *Polynima*, that is the first binary recompiler able to handle multi-threaded binaries. Through a combination of static and dynamic analysis, Polynima addresses the challenges of

lifting non-deterministic binaries. Polynima can lift binaries that use various synchronization primitives and access thread-local storage. It also ensures that the memory-ordering of reads and writes to concurrently accessed memory location is preserved in the lifted programs. Finally, we demonstrate additional applications of our recompiler and our data-flow instrumentation framework in Chapter 5.

Chapter 2

Binary Recompilation

Binary Recompilation is the process of "lifting" a program from machine-code to a compiler-level intermediate representation (IR) and, subsequently, "lowering" it back down to a new and independent executable. The IRs targeted by recompilers are typically significantly more structured than the binary representations of the programs they lift. Many of the analyses required to recover IR-level program semantics, including inference of control-flow and data structures, are of undecidable nature. Lifters can omit recovery of higher-level structures without diminishing their ability to generate semantically equivalent programs. However, most compiler-related applications, ranging from different program analyses and to any kind of reoptimization, benefit from the availability of IR-level program semantics. Since perfect lifting and recompilation is infeasible, recompilers employ various techniques to lift binaries that have varying impacts on the accuracy of recovered structures and semantic correctness of the final program.

2.1 IR-Level Program Recovery

Faithful lifting of binaries to compiler-level IRs involves extraction of IR-level program semantics within the input binary and encoding them within the higher-level representation. Binaries are usually distributed in a well-known format (such as ELF [19], Mach-O [9], or PE [38]). These formats are well documented and can be parsed unambiguously. They describe how to map the program into memory and link it with shared libraries, and provide the address at which execution starts. Compared to that, common compiler-level IRs (as implemented by GCC [20] or LLVM [29]) are significantly more structured. They commonly represent programs as a collection of global variables and functions. Functions themselves are composed of local variables, and instructions that are grouped into basic-blocks forming a control-flow graph.

Most information detailing IR-level structures is lost while initially compiling a program from its source code. Compilers can be instructed to retain descriptions of source-level structures (e.g., symbol tables, or debug information) and embed them as metadata into the generated binaries. However, software vendors usually strip this information from binaries before distributing them to end-users. Without access to this metadata, recompilers have to resort to binary program analysis techniques to recover IR-level program semantics.

Figure 2.1 provides a simplified overview outlining the structure of typical binary executables and the information recompilers typically try to extract to encode within lifted programs. Parsing the header and segment mapping of the binary yields a limited overview of the program's internal structure. The header provides the program's entry point, and the mappings can classify segments of the binary that likely contain code or data. Starting from the entry point, the lifter identifies and disassembles instructions that are reachable during the program's execution and reconstructs the program's control-flow graph. The control-flow graph is then partitioned into functions, which identify control-transfers to function entries as calls. After performing code discovery, the lifter identifies local and global variables within the program's stack and global memory, respectively.

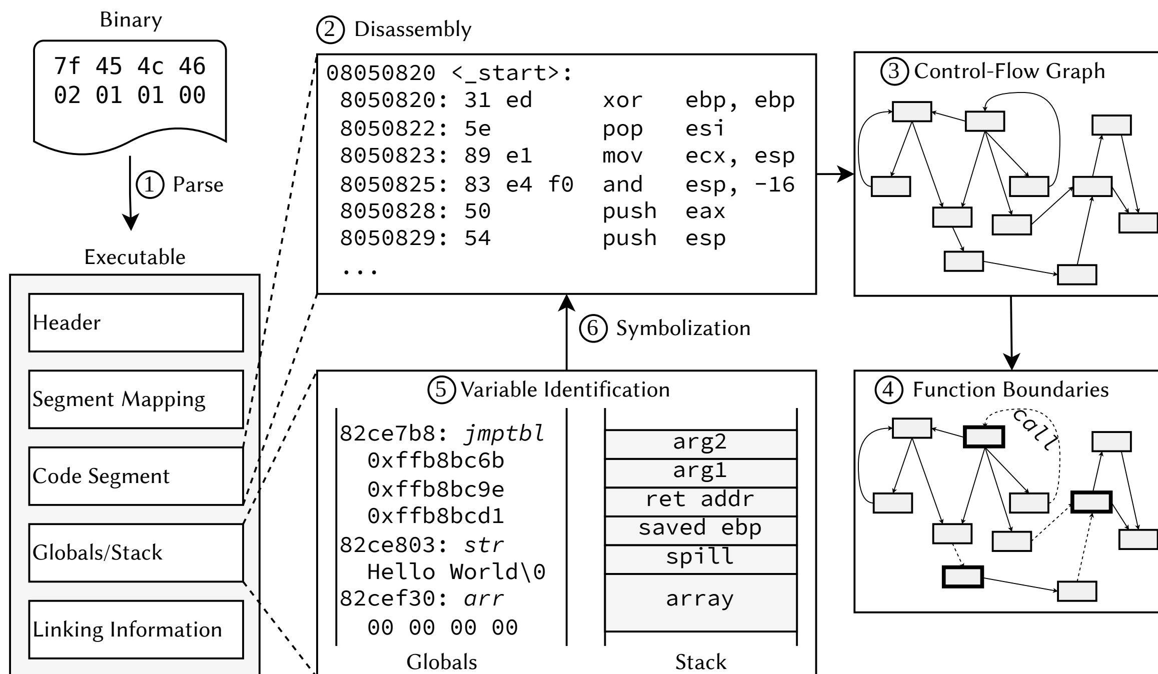

Figure 2.1: Overview of the different analysis tasks involved in lifting a binary to a compiler-level intermediate representation.

Finally, all references to these variables within the code and data segments are symbolized by replacing address calculations with symbolic references.

Although the figure depicts the different analysis tasks as isolated, they are commonly interleaved. For example, while recovering the program's control-flow, the lifter will encounter indirect control-transfers. The destination address of an indirect jump can be loaded from the global data segment by the instructions leading up to it. This pattern suggests that the control-transfer relies on a jump-table. Determining the targets of these can be intractable for the disassembler, which could lead to the omission of entire instruction sequences in the lifted program. However, if the lifter is aware of the jump table layouts generated by common compilers, it could attempt to identify the table's entries and symbolize them. Symbolization of these entries then reveals the entry points to potentially undiscovered instruction sequences, which can be processed by the disassembler to recover the additional code.

Insights gained in one analysis pass often reveal new opportunities to recover additional IR-level structures that were overlooked by previous analyses. Exploitation of these feedback effects can benefit the fidelity and quality of the lifted program. Nevertheless, the impact of different analysis techniques on lifted programs can usually be characterized independently from each other. Since the semantics of any complex target binary can only be approximated at best [25], analyses reasoning about the program's state at different program points (e.g., pointer-analysis or value-set analysis) hardly scale beyond the local scope. Limiting analysis to the local scope, recovery techniques have to account for many sources of uncertainty that could only be determined by considering all execution-paths of the program. Since this is not possible, recompilers often make assumption about program invariants that are not computable on their own (e.g., whether an address within an established basic block is reachable through an independent indirect control-transfer). This risks introduction of subtle changes into the lifted programs. However, without making any assumptions about the binary's internal structure, it is difficult to fit complex target programs effectively into the higher-level structures that make up compiler-level IRs. Finding suitable assumptions that preserve all intentional functionalities while dismissing accidental behaviors that result from code generation is a key challenge of recompilers.

2.1.1 Disassembly and Control-Flow Recovery

The ambiguity between intentional and accidental program semantics encoded in binaries can be illustrated on the example of disassembly. The purpose of disassembly is to identify and parse all machine instructions that are contained in the binary. Although Figure 2.1 suggests that the code segment is distinct from any data embedded in the binary, real binaries often contain some data within the code segment adjacent to machine instructions (e.g., jump tables and literal pools [10]). Since compilers do not insert labels into the generated binaries that annotate the start of individual instructions or sequences of instructions, it is not clear whether a byte sequence within the code segment encodes an instruction or data. Hence, it is up to the disassembler to determine for each

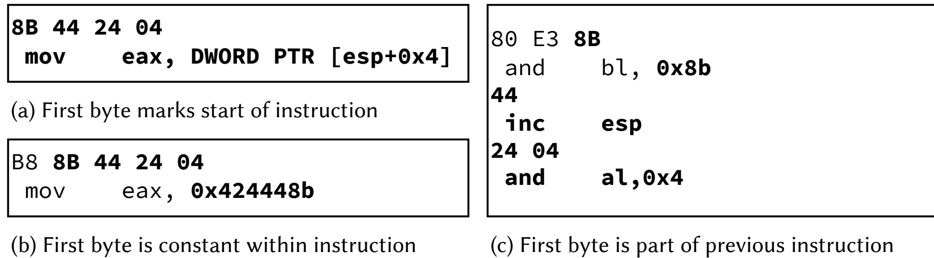

Figure 2.2: Possible interpretation of a sequence spanning four bytes within the instruction stream.

address, whether it encodes an instruction, whether it is part of another instruction, or whether it is data that is not meant to be executed.

In reality, not every address in a binary can be assigned unambiguously to a single category within this trichotomy. There are no definite distinguishing features between byte sequences that encode instructions and byte sequences that encode plain data. In fact, many random byte sequences collide with the encoding of various instructions. To make matters more complicated, on some architectures, such as x86, a byte sequence can be decoded into multiple distinct instruction sequences. Since instructions can be of varying lengths and are tightly packed with no alignment, the same byte sequence encodes different instructions depending on the starting address used by the disassembler. Figure 2.2 illustrates how the position of the disassembler within the byte stream leads to different instructions being decoded.

It is not very difficult to construct a program that intentionally branches to an address that lies in-between two instructions. Figure 2.3 shows a function containing a "hidden" jump encoded within the immediate constant of an x86 `mov` instruction.[1] If the disassembler cannot recognize or discards the branch instruction at address `0x1182`, the lifted program will be incomplete.

Embedding program semantics in overlapping instruction streams is a technique that is most commonly used in malware to conceal certain functionalities within the binary. When dealing with such binaries, any address could be part of the program's control-flow graph. One approach to fully disassemble such a program is to include every instruction sequence that can be decoded

[1] The full C source code of this example is listed in Appendix A.1.

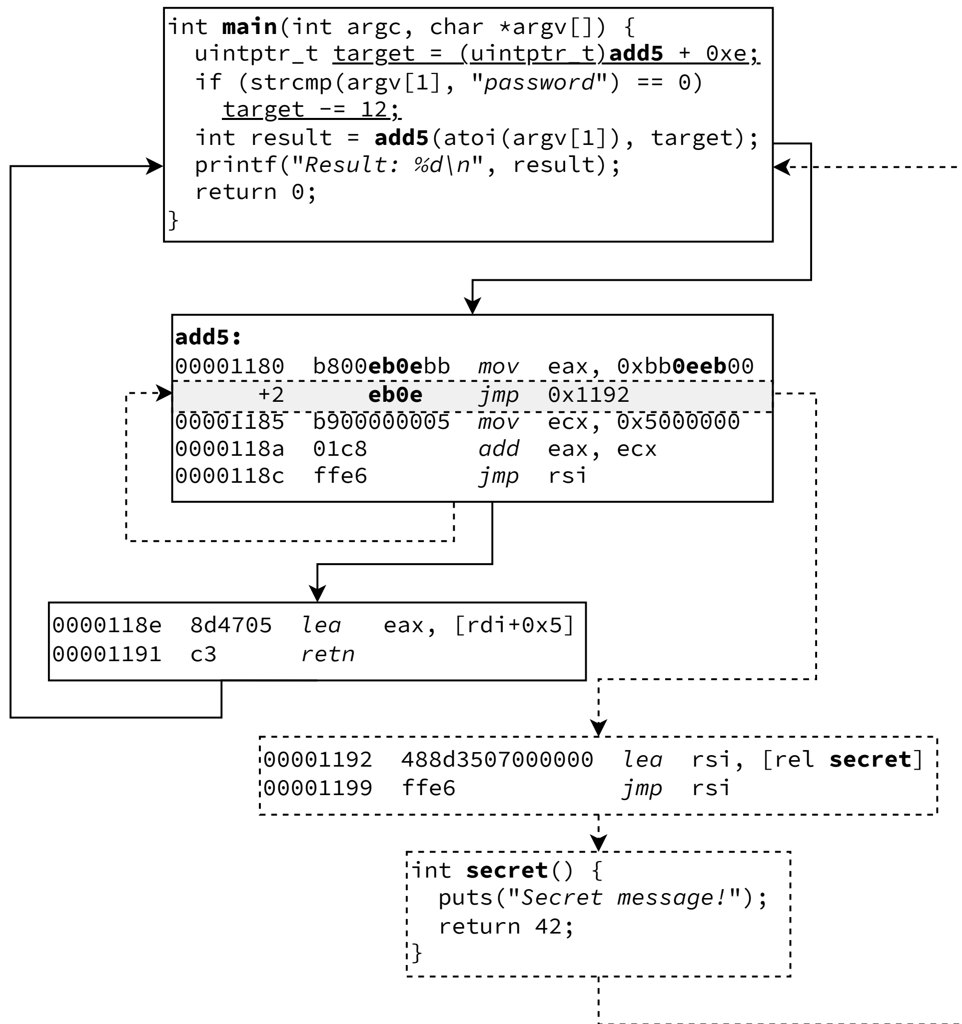

Figure 2.3: Example of an intentionally overlapping instruction sequence.

from the binary. Bauman et al. [11] have developed a tool *Multiverse* that builds on this idea. They demonstrate it is possible to disassemble arbitrary binaries, perform instruction-level modification and instrumentation, and reassemble them into executable programs while preserving every execution path. The programs Multiverse generates are effectively composed of every instruction that is contained within the binary. Although Multiverse removes all instructions that are contained in a sequence ending with an invalid opcode (which would cause a crash in the program), the text sections of the reassembled binaries are usually about 4–5 times larger when compared to the original binaries.

Including every instruction sequence in the lifted program is rarely necessary for achieving high coverage. Apart from the fact that these programs are significantly larger, including "extra functionality" not part of the original binary makes it more difficult to reason about the program's invariants and semantics. For example, overlapping sequences perform different register assignments and often end in the same jump instruction. Hence, the target of such a jump has additional predecessors with different register states. This makes it difficult to reason about registers values and hinders any kind of analysis that requires knowledge of such values.

In order to extract a more meaningful set of instructions from a binary, disassemblers are usually equipped with strategies to discard instructions that are unlikely to be part of the program's original machine code. With Multiverse, this strategy would state to discard nothing. However, many binaries of interest for recompilation are unobfuscated *commercial off-the-shelf* (COTS) programs that were compiled from their source code with a standard compiler, such as GCC, Clang, or MSVC. Unlike the example in Figure 2.3, these programs contain no intentional control-transfers to addresses that lie in-between instructions on all *regular* execution paths. That does not rule out there are inputs to the program that divert control-flow to such addresses (e.g., a stack-buffer overflow overwriting a function's return address). However, such inputs are not part of the program's intended semantics and are not required to be supported by the lifted program. In fact, the behaviors triggered by such inputs are usually undesired and their presence is often a sign of potential security vulnerabilities (e.g., return-into-libc [45]). Hence, if it is known that the target binary is a regular COTS program, any overlapping instruction streams can be discarded.

This assumption significantly simplifies the disassembly process. Nevertheless, identifying the start of every valid instruction sequence is a non-trivial task. As outlined above, binaries often contain data within the text segment that is not meant to be executed. If the disassembler mistakenly interprets some of that data as instructions, it can produce instruction sequences that overlap with real instructions. If overlapping sequences are automatically discarded, accidental disassembly of data within the text segment can cause the resulting program to be incomplete.

To identify instructions that are likely reachable during actual program executions, recompilers like McSema [48], Rev.ng [17], mctoll [53], and SecondWrite [6], but also commercial binary analysis tools like Ghidra [39] or IDA Pro [23] combine disassembly with control-flow recovery. A program's control-flow graph is a directed graph that represents all control-transfers that can occur during the program's execution. Since most instruction sequences are target of at least one direct jump, an indirect jump through a parsable jump-table, a call, or a return, the disassembler can "just" explore the program's control-flow graph to discover valid instruction sequences, starting from the program's known entry point. Although this approach works well to recover the majority of instructions in many binaries, it does not guarantee to locate all functions and blocks that are reachable through indirect control-transfers. It also does not guarantee to identify all targets of individual indirect control-transfers, even if the targets themselves have been successfully discovered through other paths.

State-of-the-art analysis tools employ various heuristics that can decode many types of indirect control transfers. There is a plethora of techniques to decode jump tables and to detect functions that are only reachable through indirect calls [40]. Among other techniques, these approaches employ heuristics like scanning for C++ vtables, parsing exception handling and unwinding information, scanning for function pointers in the data segment, or performing pattern matching on instructions that resemble function entries. Unfortunately, all these heuristics are best-effort. Despite investing decades of research into optimizing these heuristics, lifters that purely rely on static analysis techniques are prone to introduce errors in the lifted programs unexpected ways [32]. Even worse, heuristics might need to be adjusted to recognize patterns in binaries of the same programs that were generated by different compilers or with different optimization levels [7].

One strategy that achieves a coverage that is conservative (i.e., avoids false positive instructions) while capturing sound (but not complete) program semantics is to fully rely on traces of the

program's execution. Through observations of the actual control-flow of the program, the disassembler can record the addresses of all executed instructions and the targets of all indirect control-transfers. While this approach cannot capture a program's entire semantics, the result is sound in the sense that it accurately captures the semantics of the program paths that are relevant for the provided inputs. In fact, some argue that this incompleteness can even be a feature, since it effectively "debloats" the target program and reduces its attackable surface by removing all semantics that are unnecessary to process the provided inputs. Since every instruction that is not part of the recorded traces can be discarded, and the targets of all indirect control-transfers are known, this approach enables reliable lifting and recompilation of programs that are fully *deterministic* (i.e., they are guaranteed to take the same paths in every execution of the same input). Altinay et al. [3] have implemented this approach in BinRec and demonstrate it to be effective in lifting and recompiling a variety of single-threaded COTS programs.

Naturally, execution traces can be incomplete for a specific input when the target programs are not fully deterministic. This is the case for most multi-threaded programs. It also affects single-threaded programs that contain control transfers based on state that is intentionally non-deterministic across executions (e.g., pointer-values/ASLR, or randomly generated numbers). Although this can be mitigated partially by tracing the program repeatedly, the resulting traces might still be incomplete. However, Altinay et al. argue that failures during execution caused by incomplete traces can be reliably observed during program execution. If the recovered binary is augmented to recognize such missing paths in the control-flow graph, they can be incrementally added to the lifted program [3].

Note that this discussion omits any programs that generate and modify code at runtime. This commonly occurs in just-in-time compilers (JITs) and is also used as a technique to conceal malicious code within malware. It is worth highlighting the special ambiguity between code and data in these programs. Since code is generated at runtime, it is written to memory in the same way as data is, but is then executed as code. For these programs, data and code are not distinct. For

binary lifting, it might be in the operator's interest to lift the code concealed within the malware, while she might not intend to lift the code that is generated by a JIT engine. In this work, it is assumed that the target programs are not generated or modified at runtime.

2.1.2 Function Boundary Detection

Functions are a key building block of any program to facilitate modularization and code reuse. They structure the program in two important ways. First, they constrain the set of targets of all return instructions in the program's control-flow graph. Returning from a function resembles an indirect jump that is guaranteed to transfer control back to the address exactly after the call used to enter the function. Without the additional structure that functions provide for the control-flow graph, all return instructions have to be treated as indirect jumps that return to any of its function's call sites. Figure 2.4 illustrates this difference. Without function boundaries, the control-flow graph suggests that a call from `main` to `f1` could return to both `main` or `f2`. If functions are present, execution always continues at the instruction after a call.

This property is crucial for many analyses and optimizations on lifted programs, since it drastically simplifies the control-flow graph. In particular, note that the control-flow graph on the left contains a cycle hinting at a potential loop, while the control-flow graph on the right does not. It also makes it possible to reason about functions in isolation, rather than having to consider the entire program's control-flow graph.

The second benefit of structuring a program through its functions is that it unveils how the code interacts with stack memory. When recovering functions, the lifetime of function-local objects can easily be expressed in terms of the control-flow graph. These objects are typically allocated on the *stack*, which is a contiguous region of memory allocated by the operating system for each thread. The stack is accessed through the *stack-pointer*, which is a register that points to the top of the stack. Upon function entry, the program typically adjusts the stack-pointer to

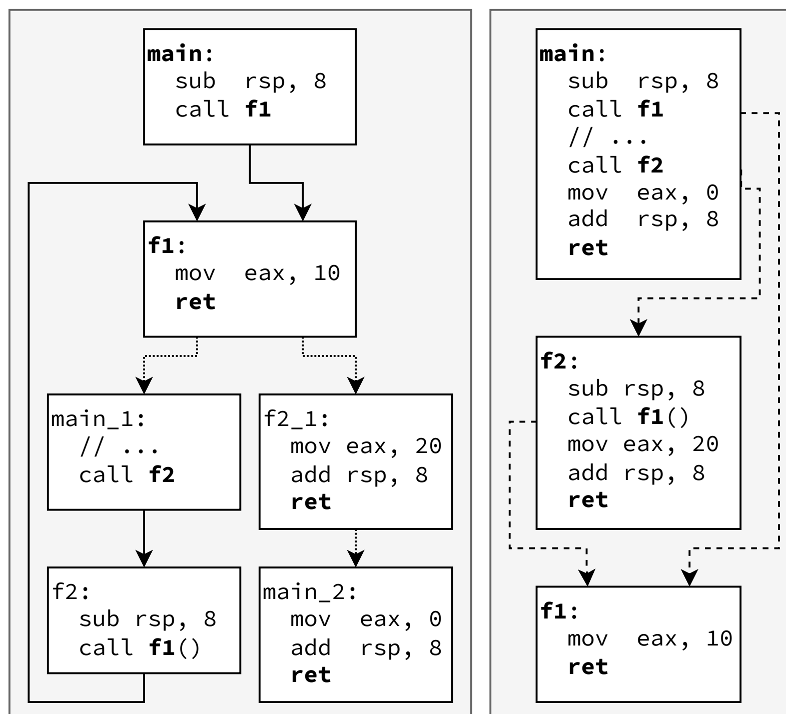

Figure 2.4: Example of a program's control flow graph without functions (left) and with functions (right).

allocate space for local variables and spills. The moment the function returns, the stack-pointer is adjusted back to its original value, which effectively releases the function's local variables. The boundary between two frames on the stack is *always* determined by the value of the stack-pointer on function entry [37]. This value is also known as the *frame-pointer* and is typically stored in a dedicated register, but most compilers also offer an option to omit it for most functions. The space between two frame-pointers is typically referred to as a function's *stack-frame*. Every function can only access local variables that are part of its own frame, or arguments that are placed at the end of its caller's frame. This is not explicitly enforced by the hardware. However, any direct access outside the executing function's stack frame comprises undefined behavior. Since the frame-pointer serves as a reference to allocate any local variables, local variables can be identified by its owning function and its offset from the frame-pointer. This property is important when recovering local variables (see Section 2.1.3).

Detection of function boundaries is a non-trivial task. One central approach promising good results is to consider the targets of all call instructions. The targets of these instructions are marked as function entries, and all blocks that are reachable from these entries and do not end in a return instruction are part of the function. There are several challenges to make this approach work. Naturally, it requires that the disassembler recognizes call instructions in the first place, and that there are no false positive call instructions with targets that lie within functions. Success of such an analysis also depends on the control-flow graph's completeness and its accuracy in resolving targets of indirect calls, since not all functions are reachable through direct calls.

Putting disassembler- and CFG-related concerns aside, relying on special instruction to indicate function boundaries is not reliable in all cases. Instead of using call and return instructions, compilers are at liberty to use other mechanisms to transfer control between functions; calls can be implemented by combining a push with a jump instruction, and returns can be implemented through a pop and a jump instruction. Without these markers, function detection becomes much more difficult, and is out of scope of this discussion.

Fortunately, the COTS programs are usually compiled such that they use call and return instructions to implement *most* function calls. Nevertheless, even in COTS binaries, a target of a call instruction is not necessarily a function entry, not all calls to function entries are dispatched through call instructions, and not every function exit is encoded through a return instruction. For example, on some architectures, such as 32-bit x86, there is no facility to access the value of the register holding the current instruction pointer directly. This value is needed to compute the absolute address of global variables in position-independent code. One way to obtain the value of the instruction pointer on x86 is to issue a call instruction to the instruction immediately following the call (i.e., `call 5`, since this instruction has a length of 5 bytes). This instruction pushes the address of the instruction following the call onto the stack, which is then moved into a register by a successive `pop` instruction.

The other common instances of control-transfers between functions that are not encoded through call instructions are *tail calls*. When a function's last statement is a call to another function, the compiler can, in certain cases, jump directly to the target function, instead of issuing a call that is followed by a return instruction. Since this jump is at the "tail" of the calling function, it is called a tail call. These calls are not an issue, if the callee is known to be a function entry by means of a different call. Some functions are only reachable through tail calls, however. This can confuse some tools, like Nucleus [8], which would merge the callers and callees into a single function with multiple entry points.

Finally, not every function-call returns to its call-site. For example, the function `error (int status, int errnum, const char *format, ...)` terminates the program if the first argument is non-zero [34]. If the compiler can prove (or the developer can assert) that the first argument at a particular call-site is always non-zero, it can safely omit any instructions following the call to `error`. This can be a problem for binary analysis, if the called function's non-returning behavior is unknown, or the range of values for the first argument cannot be determined. In the worst case, the non-returning is call placed at the end of the function, such that it immediately preceeds the entry of another function, which can cause the two functions to be merged into a single function [8].

Apart from calls and returns, there are other mechanisms to facilitate control-flow between functions, such as exceptions, and `setjmp/longjmp`. In regular COTS programs, these mechanisms are usually layered on top of functions and thus have only minimal impact on the partitioning of program's control-flow graph into functions. In fact, exception and unwinding tables can be useful for CFG recovery and function boundary detection [5]. Furthermore, if function boundary detection is for some reason not possible, it is perfectly possible to lift a program to a compiler-level IR and recompile it into a new binary without identifying any function boundaries at all [3]. However, the recovered program would suffer from the previously mentioned drawbacks, and recovery of local variables is likely not feasible.

2.1.3 Variable Identification and Symbolization

Having recovered the program's instructions and control-flow, the next step is to recover the variables and objects of the program. The primary purpose of this step is to replace direct references to global or stack-allocated memory with symbolic references throughout the program. There are two major tasks involved in this process. The first is to identify the set of variables and objects that are contained within the program's global and stack-memory. The second is to locate all references to these variables throughout the program's code and data, and to replace these references with symbols that denote the respective recovered variables. Since heap-objects are allocated and managed by the standard library, there is no need to symbolize them.

The significance of symbolizing variables is that it expresses the program in terms of the variables, rather than opaque memory addresses. If symbolization is omitted, all accesses to global and local variables have to be expressed by their addresses in the original program. This means the layout of any data in the original binary has to be replicated byte-by-byte in the recompiled program. For example, if the original program contains global variables `x` and `y` at addresses `0x08048100` and `0x08048104`, the recompiler has no choice but to allocate the same two variables at exactly these addresses in the lifted program. This makes it infeasible to transform the program effectively, such as changing the size of any variables (e.g., extending the size of a timestamp to 64-bit), or modifying the layout of the program's data (e.g., inserting red-zones between variables to retrofit the binary with AddressSanitizer [43]). Furthermore, it is detrimental to the compiler's ability to determine whether a memory access affects a particular variable. Without partitioning memory into distinct variables, there is no notion of variable bounds. Hence, the compiler has to assume that most memory accesses through a computed address might overwrite any adjacent variable values (i.e., all other variables within the whole global segment or the same stack frame), even if the pointer was derived from a known base address.

Similar to instructions, stripped binaries contain no metadata that can be used by the lifter to identify boundaries between different variables or locate direct references to them. The primary source of information to infer this information is analysis of how the program interacts with memory. Intuitively, the program's memory can be partitioned into distinct variables by identifying direct references to global or local memory (i.e., *base pointers*), and determining the range of pointers derived from these base pointers. Any base pointers with overlapping ranges are part of the same variable. By coalescing these ranges, a lifter can determine the size and the layout of variables in the binary.

Base pointers to variables are usually computed in one of the following ways:

1. In position-dependent binaries, global variables are referenced by their absolute addresses.
2. In position-independent binaries, the absolute address of global variables is not known at compile time, since the program is placed at a different address for every execution. Because the distance between global variables and the instructions that use them is constant, they can be addressed by adding an offset to the program counter.
3. Addresses for local variables are computed by adding an offset to the stack-pointer or the frame-pointer registers.

While base-pointers derived from registers are relatively easy to identify, processing absolute addresses is more difficult. In binaries, pointers and integers share the same representation. If an integer value collides with the address of a global variable, it can be very difficult to determine whether the value is used as a pointer to the variable, or as an integer. The only way to determine whether a value is a pointer to a variable requires observation of how it is used. However, this might not always be trivially possible. Consider the example in Listing 2.1. The function `f2` stores the address of the global variable `arr` in a union that is also used to store an integer. Function `f1` stores an integer that collides with the address of `arr` in the same union. After compiling the

program, the functions `f1` and `f2` generate identical code in x86. Functions `f3` and `f4` then access the value of the union as an integer and as a pointer, respectively. To determine whether the values stored by `f1` and `f2` are pointers to `arr`, or integers, the lifter has to reconstruct the data-flow of these values with high precision. Only the value that is used as a pointer must be symbolized as a reference to `arr`, whereas the value that is used as an integer must remain unchanged.

Listing 2.1: Collision of a constant with the address of a global variable. In x86, both functions generate identical code.

```
1  union { long x; int* p; } u;
2  extern int arr[10];               // &arr[0] == 0x08048100
3  extern int n;
4  void f1() { u.x = 0x08048100; } // mov dword u, 0x08048100
5  void f2() { u.p = arr; }        // mov dword u, 0x08048100
6  void f3() {
7      long x = u.x;               // mov eax, dword u
8      printf("%ld\n", x);
9  }
10 void f4() {
11     int* p = u.p;               // mov eax, dword u
12     int k = n;                  // mov edx, dword n
13     int val = p[n];             // mov eax, dword [eax+edx*4]
14     printf("%ld\n", val);
15 }
```

Once base pointers are identified, the lifter determines the range of pointers that are derived from them. The upper and lower bounds of these ranges determine the size of the object. As with pointer-identification, it requires precise knowledge about the dataflow of the pointers. For example, the base-pointer computed by function `f2` in Listing 2.1 is used in function `f4` to access an arbitrary element of the array. The index of the accessed element is controlled by another global variable `n`. To infer the size of the array, the lifter has not only to connect the use of the base-pointer in `f4` to the computation of the base-pointer in `f2`, but also to determine the range of values that `n` can take on. Depending on the usage of `n` throughout the program, computing a precise range for `n` might not be possible. Here, the lifter has to estimate the range of `n`, or rely on other sources of information to determine the size of the array.

Listing 2.2: Chunk-wise initialization of array that is then passed to an indirectly called function.

```
void (*fptr)(int*);
void init(int*, size_t len);
void f() {
    int arr1[12];       // &arr1[0] == ebp-0x60
    int arr2[4];        // &arr2[0] == ebp-0x30
    init(arr1 + 0, 4);
    init(arr1 + 4, 4);
    init(arr1 + 8, 4);
    init(arr2 + 0, 4);
    fptr(arr1 + 4);
}
```

Symbolizing variables without considering all their uses can introduce errors into the lifted program. Consider the example in Listing 2.2. The function `f` initializes its arrays in chunks of size 4 and then passes a pointer to the second chunk of `arr1` to an indirectly called function. The arrays are located adjacent to each other on the stack. Assume that the targets of the indirect call are not known to the lifter. Hence, it is not known which elements of `arr1` are accessed by the indirectly called function. If the lifter considers only the information provided by the calls to `init`, it seems like there are 4 separate arrays on the stack. If it ignores the call to `fptr`, symbolization will incorrectly split `arr1` into 3 separate arrays. This might cause out-of-bounds accesses when calling `fptr`. In order to not break the program, it is necessary to symbolize `arr1` as a single array. Since it is not clear whether the adjacent `arr2` is part of the same array, a conservative symbolization approach would have to merge it with `arr1`. Accounting for unknown uses of variables is a key challenge of variable identification and symbolization. As every reasonably complex program contains objects that are indirectly accessed and that static analyses cannot reason about, lifters are forced to operate conservatively to avoid introducing errors into the lifted program. This means that stack-allocated arrays and objects usually cause the whole stack-frame to be symbolized as a single variable (i.e., not recovering individual variables at all), and that global variables are not symbolized at all [6].

Note that this step does not infer nominal or structural types of any variables. Type information is very important for decompilation, but has little impact on most compiler-level analyses and optimizations. Nominal type information provided by front-ends, such as Clang, can inform the compiler about pointers that cannot alias. In LLVM for example, this information is not directly encoded in the IR, but is layered on top as metadata [33]. Recovery of nominal types to inform the compiler about pointer-aliasing is beyond the scope of this work. Recovery of structural types will not be discussed either. Structures types within compiler-level IRs are primarily used as an aid to express pointer-computation of nested arrays and structures. They can be omitted in favor of byte-level offsets, which suffice to express the same pointer-computation.

2.2 Preservation of Program Semantics

The previous sections outlined the various tasks and challenges involved in recovering IR-level program semantics from a binary. Since the required analyses are subject to uncertainty, the recovered program can not always be perfectly mapped onto compiler-level IRs. To produce semantic-preserving binaries, recompilers have developed various techniques to mitigate the impact of these uncertainties on the lifted programs. This section outlines the most common techniques. Note that usage of the following techniques establishes a baseline to embed arbitrary programs into compiler-level IRs. In many cases, recompilers can eliminate uncertainty from large parts of the lifted programs and utilize this knowledge to improve the program's IR without loss of semantics. Nevertheless, since there will always be some imprecision, discussion of the following techniques is necessary to understand the anatomy of lifted programs.

2.2.1 Instruction Emulation

At their core, every lifter has to translate arbitrary sequences of machine instructions into semantically equivalent sequences of instructions expressed in the target IR. Modern instruction set architectures (ISAs) comprise hundreds of instructions. Many of these instructions modify multiple registers simultaneously, operate on processor-internal state, and/or are implicitly affected by status or control registers. Mapping these instructions on a higher-level IR is not always straightforward. For example, the x86 `EFLAGS` register is one of the most important status registers. It is implicitly changed by many instructions (that all modify different flags), and its value is used to determine the behavior of other instructions (e.g., conditional jumps). To translate an x86 instruction that depends on the value of `EFLAGS`, the lifter has to keep track of the value of `EFLAGS` throughout the program's execution. It can often be tricky to determine the value of `EFLAGS` at a particular program point, since its state accumulates across instruction sequences that might not be directly related to the instruction being translated. This is a common problem when lifting any code depending on processor-internal state, and is not limited to x86.

The solution to this problem is astoundingly simple: the lifter generates a new program that effectively emulates the original binary instruction-by-instruction on a virtual CPU. This virtual CPU is allocated as a global variable, and it replicates the processor's state (e.g., general-purpose registers, status registers, control registers, or the x87 floating-point stack) at every program point. The lifter then translates the original binary's instructions into sequences of instructions that manipulate the virtual CPU's state. For instance, a 32-bit x86 `push` instruction is lifted to an instruction-sequence, that (1) subtracts 4 from the emulated `esp` register, (2) loads the pushed operand, (3) stores the operand to the address in `esp`, and (4) sets the emulated program counter to the next instruction's address. Although this technique initially increases the size of the program significantly, most generated instructions are redundant and can be eliminated. For example, the virtual program counter is updated within every translated instruction-sequence, but is only used to compute addresses of global variables or determining the targets of indirect control flows.

Despite the simplicity of this approach, it is very effective in translating sequences of machine instructions into a different representation.

Implementing a CPU-emulator that can emulate large subsets of modern ISAs is a non-trivial endeavor. Some recompilers bring their own translation layers, such as McSema [48] (through Remill [49]) or SecondWrite [6]. Another option is to use existing battle-tested CPU-emulators. In particular, the processor emulator QEMU [13] has proven itself as a useful foundation for this style of lifting. To emulate programs from arbitrary architectures, QEMU uses its *Tine Code Generator* (TCG) to translate machine code through architecture-specific frontends to its own architecture-independent IR. The operations used by the TCG IR directly manipulate a virtual CPU and can be easily translated to compiler-level IRs. This makes it an ideal candidate for recompilation and is used as an intermediate step for lifting x86 programs to LLVM IR by BinRec [3] and RevGen [15] (which both build on S2E's translator [16]), and Rev.ng [17].

Programs that were lifted using this method require some additional setup to run. Assuming that references to global variables cannot be reliably symbolized, the original binary has to be embedded at its exact loading address into the image of the lifted binary. This keeps any addresses computed by the virtual CPUs consistent with the original binary's memory layout. Similarly, references to stack variables are computed through the virtual CPU's stack-pointer. Hence, the stack has to have the same layout as in the original binary. However, the lifted binary requires additional stack-space in each function (e.g., to store return addresses and spilled registers). The fix is to embed two stacks into the recompiled binary: one that is used by the lifted program, and one that is used by the virtual CPU. Figure 2.5 illustrates the anatomy of a recompiled binary that was lifted using this technique.

Mapping the original binary into the address-space of the lifted program and allocating a separate stack for local variables circumvents the need to symbolize any references to data. However, computed addresses that are used as jump or call-targets within indirect control-flows still point to the original binary's text section. Unlike data references that are explicitly preserved from the

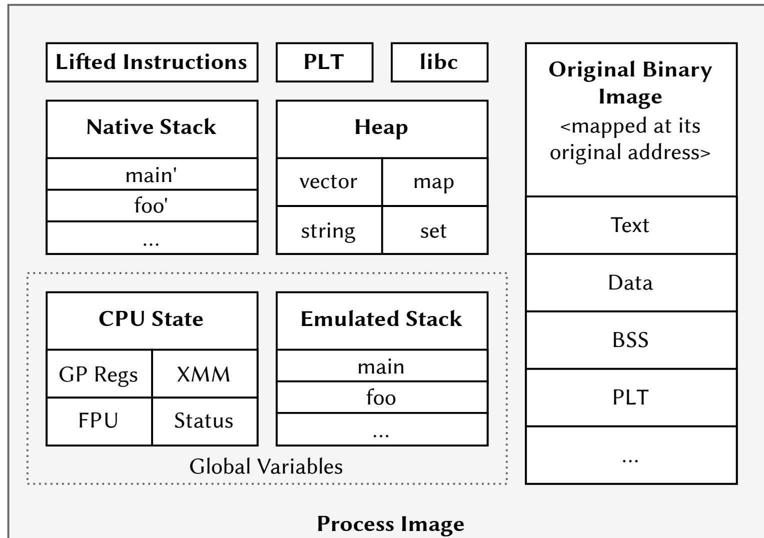

Figure 2.5: Process image of a recompiled binary.

original binary and are reused in the lifted binary, reusing code from the original text segment is undesirable. Since the layout and placement of functions and blocks within the recompiled binary is difficult to control, addresses from the original binary's text segment cannot be mapped linearly onto the recompiled binary's text segment. Instead, the recompiled binaries map addresses to the original binary's text segment to the corresponding lifted functions and blocks using a lookup-table at runtime. If the targets of the indirect control-flows are not known at compile-time, the lookup-table is best implemented as a central dispatcher that is called by every indirect control-flow and maps every target-address. This dispatcher tends to be very large and adds significant overhead to the lifted program. Since every basic block is preceded by this dispatcher, the whole program becomes impervious to any kind of analysis spanning multiple blocks. If functions are recovered, the scope of the dispatcher can be reduced significantly. Rather than having a single dispatcher for the whole program, the lifted program can have a dispatcher for every function that requires it [22]. This approach is more efficient, but still adds many false-positive edges to the control-flow graph.

If the program's control-flow graph can be recovered, such that the targets of each indirect control-flow are known, the dispatcher can (and should) be omitted. This is the case when the program's

control-flow graph is recovered through execution traces, as described in Section 2.1.2. Here, all indirect control-transfers (including indirect function calls) can be replaced with a switch statement that lists only valid targets. Depending on the size of the switch statement, the compiler can optimize it to a jump-table, or a series of conditional jumps. On paper, this technique requires more instructions than the indirect jump in the original program. However, it turns every indirect control-flow into a direct control-flow with a data dependency. Knowing the (usually rather small) set of targets to an indirect control-transfer is highly beneficial to the recompiler's ability to analyze and also optimize the lifted program. And while some of the valid targets of the indirect control-flows might not be part of the lifted program, it acts as a CFI mechanism that is capable of detecting any control-transfer that was not observed during tracing.

2.2.2 CPU Switching

The previous section outlined how to map arbitrary programs onto compiler-level IRs by emulating the original binary's instructions on a virtual CPU. Most binaries rely on dynamically linked libraries for a variety of functionality, such as memory allocation, file I/O, or network communication. These libraries are not part of the original binary and are usually not lifted by the recompiler. Hence, the machine code provided by these libraries is not executed on the virtual CPU.

Arguments to functions in these libraries are usually passed through a combination of registers and the stack. In the lifted program, the arguments to the external functions are placed accordingly in the registers of the emulated CPU and at the top of the emulated stack. The external call is not aware of the virtual CPU, however, and it cannot access the arguments that are passed to it without additional support. For many external functions, this is not a problem, because their signatures are known. The recompiler can use this knowledge to generate supporting code that loads the arguments from the virtual CPU according to the calling convention of the target architecture,

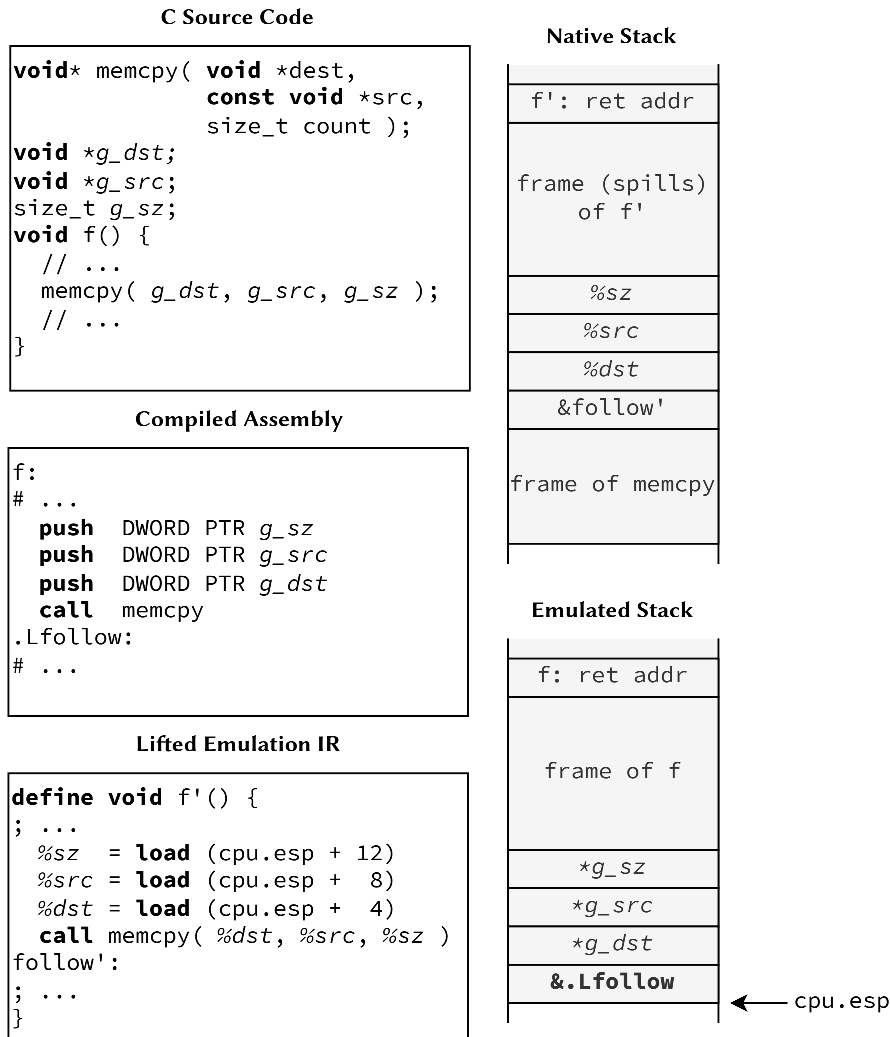

Figure 2.6: Lifted IR for a call to `memcpy` on 32-bit assembly.

and uses these values to call the external function. Consider the example in Figure 2.6. All three arguments of a call to `memcpy` on 32-bit x86 are pushed in reverse order onto the stack immediately preceding the call. These pushes are emulated in the lifted program, and the arguments are placed at the top of the emulated stack. The lifted code then can load these arguments from the emulated stack and pass them as arguments to `memcpy`. The stack-frame of `memcpy` is allocated on the native stack, and the emulated stack is not used until the function returns.

The situation is more complicated when the arguments of a call cannot be determined by the recompiler. For example, the signatures of the functions provided through proprietary linked

libraries could be unknown, or the external functions are variadic. In these cases, the recompiler has not enough information to generate the supporting code that loads the arguments from the virtual CPU and stack to call the external function. The solution to this problem is a technique called *CPU switching*. For the duration of the external call, the native CPU assumes the state of the virtual CPU to execute the library function. The lifted program facilitates this by deserializing and serializing the state of the virtual CPU onto the native CPU immediately before and after the call to the external function, respectively. Since the lifted program prepares the emulated CPU and stack to match the calling convention for the target function, the external function can access all its arguments and return values as if it was called by the original program. Unlike the lifted `memcpy` call in Figure 2.6, the stack-frame of the external function is allocated on the emulated stack, and the native stack is not used until the CPU is deserialized.

Compiler-level IRs have no notion of CPU-switching (in particular, replacing the stack-pointer with a runtime-provided pointer), which means the recompiler has to provide a mechanism to deserialize and serialize the CPU. This mechanism can be implemented using inline assembly that explicitly assigns registers from the virtual CPU to the native CPU, and vice versa. Usually, the calling convention for the affected external calls is known. The lifted program only needs to serialize the registers that are used to pass arguments to the external function, and deserialize the registers that are used to return values from the external function.

2.2.3 Load-/Store-Ordering

One aspect of binary lifting that has received comparatively little attention is the impact of implicit memory ordering in the binary on the correctness of the lifted programs. The x86 architecture, for example, provides a strong ordering guarantee called *Total Store Ordering* (TSO) for memory operations. TSO guarantees that the order in which memory writes are scheduled in the binary, and hence issued by the CPU, is the same as the order in which they are observed by other

CPUs [44]. This means that many memory accesses to concurrently shared memory locations are not explicitly designated as atomic in the binary, but are implicitly atomic because of the TSO guarantee. For example, releasing a spin-lock can be as simple as writing a zero to the lock's address. On x86, this can be implemented through a regular non-atomic `mov` instruction (if using acquire-release memory ordering). The CPU, however, guarantees that the write is atomic, and that it is correctly synchronized with other cores waiting to acquire the lock.

If the lifter cannot identify the original program's synchronization mechanisms, the lifted program can lose the original program's synchronization guarantees, and contain unintentional data-races for correctly synchronized programs. Consider the example in Listing 2.3. This code awaits another thread to signal safe access to `shared_data`. When the condition variable is signaled, the thread increments `shared_data` and resets the condition variable. The generated assembly for the access to `shared_data` is shown in Listing 2.4. Note that neither the load nor the store to `signal` uses atomic instructions. If the load of the spin-loop is lifted as a regular load in the compiler-IR, the compiler might remove the entire spin-loop (i.e., everything between the labels `.L2` and `.L3`), since it does not contain any synchronization mechanisms or other side-effects that would prevent the loop from being optimized away. The recompiled program would then access `shared_data` without synchronization, resulting in a data-race.

Listing 2.3: Access to a shared variable that waits for another thread to signal safe access.

```
1 extern std::atomic<bool> signal;
2 extern int shared_data;
3 void thread_func() {
4     while (!signal.load(std::memory_order_acquire));
5     shared_data += 1;
6     signal.store(false, std::memory_order_release);
7 }
```

Without a deep understanding of the original program's internal synchronization mechanisms, it is difficult, if not impossible, to determine whether a load or store instruction can ever access

Listing 2.4: Compiled x86 assembly for the access to `shared_data` in Listing 2.3.

```
1  .L2:
2      movzx eax, BYTE PTR signal         # Load signal value
3      test al, al                        # Check if set
4      je .L2                             # Jump back if signal not set
5  # -------------------------------------------------------
6  .L3:
7      add dword ptr [shared_data], 1     # Access Shared Data
8      mov dword ptr [lock], 0            # Reset Signal
9      ret
```

shared memory concurrently. The lifter can only assume that every memory access is concurrent, and has to insert memory fences around every load and store to ensure that the lifted program is correctly synchronized. Such an approach has been implemented by Rocha et al. in their binary lifter *Lasagne* [42]. This conservative fence-insertion can have a detrimental impact on the runtime-performance of the lifted programs. The fence-instructions themselves are expensive, and they introduce unnecessary serialization points that limit the CPU's ability to execute instructions out-of-order. Additionally, they prevent any optimizations that require reordering of memory operations (e.g., hoisting loads out of loops). Fences can only be omitted if the recompiler can prove that the memory operations are not concurrent. Lasagne uses a simple heuristic that omit fence insertion around memory access to the stack, assuming that no stack-allocated variables are shared between threads. This can be a reasonable assumption for a subset of programs, but it is not always true. To support this optimization for arbitrary programs, the recompiler requires access to precise data-flow and pointer-analyses. However, these kinds of analyses are difficult to perform for programs that were lifted using the technique described in Section 2.2.1, since they have no symbols denoting individual variables [32].

Chapter 3

Wytiwyg—Dynamic Stack-Layout Recovery for Binary Recompilation

Chapter 2 introduced the problem of identifying and symbolizing stack variables in lifted binaries and outlined the challenges that arise when attempting to solve it. Recovery of stack variables is crucial to effective recompilation, because the set of values used by any computation at any program point usually resides within the currently executing function's stack frame. Since direct pointers to local variables have no nominal types that inform the lifter about the size of the underlying objects, the lifter has to infer the stack-layout of each function based on how pointers to stack-variables are used throughout the whole program. This means that identifying variables in binaries is a data-flow problem.

This chapter introduces *Wytiwyg* ("What you trace is what you get"), a dynamic and incremental approach to identify and symbolize local variables in lifted COTS-binaries. To facilitate this, Wytiwyg employs an instrumentation-based approach that tracks pointers to stack variables throughout the program and observes how the program derives new pointers from existing ones. Unlike static approaches, relying on dynamic observation of real executions allows us to symbolize

functions with high precision while preserving all semantics that are exhibited by the traced inputs.

3.1 The Emulated Stack Analysis Hazard

Section 2.2.1 introduced the notion of an emulated stack. The overall relationship between the native stack and the emulated stack is illustrated in Figure 2.6. Both stacks are maintained in parallel and allocate a stack frame for each function invocation. The primary difference is that the emulated stack maintains local variables of the original program, whereas the native stack primarily holds saved registers and spilled variables that are artifacts of the instruction emulation.

Tracking two stacks and setting up two frames for every function call naturally has some overhead. Specifically, the lifted program emulates saved registers and spilled values of the original binary, but also has its own spills because of the increased register pressure that comes with keeping track of two stacks. However, unlike values spilled to the native stack, the compiler cannot reliably keep track of values written to the emulated stack, since the emulated stack is allocated as an opaque byte array.

Listing 3.1: Simple function with multiple indirect stack accesses.

```
1  typedef struct {
2      int x;
3      int y;
4  } p;
5
6  // Return a value x with 0 <= x < sz.
7  size_t f3(size_t sz);
8  // Return p1 or p2.
9  p* f2(p* p1, p* p2);
10
11 void f1() {
12     p *ptr;
13     p a;
14     p b[3];
```

```
15     a.x = 3;
16     a.y = 4;
17     ptr = f2(&a, b);
18     b[f3(sizeof(b))] = a;
19     ptr->y = b[1].x;
20 }
```

Consider the example in Listing 3.1 and its corresponding assembly in Listing 3.2. The stack frame of `f1` is illustrated in Figure 3.1. In the C code, `ptr` is assigned the return value of `f2` and then used to write to `ptr->y`. In the generated assembly, the return value of `f2` is spilled to the stack in line 21 and then loaded into the register `eax` in line 37 before it is used to write to `ptr->y`. Since out-of-bound accesses are undefined behaviour, the source-compiler can assume that the access to `b` in line 18 cannot overwrite the spilled value of `ptr`. This information is lost during compilation. Now, in order to infer that the loaded value of `ptr` in line 19 is identical to the spilled value in line 17, the lifter has to prove that the access in line 29 through the computed address `ebp + f3(24)*8 - 44` cannot overwrite the location that holds the spilled pointer at `ebp - 12`. Depending on the complexity of the operations involved in the address computation, this analysis quickly becomes challenging.

Listing 3.2: Generated x86 assembly of the function in Listing 3.1.

```
1  f1:
2      # save caller's frame pointer
3      push   ebp
4      # initializer frame pointer
5      mov    ebp, esp
6      # allocate stack frame
7      sub    esp, 64
8      # a.x = 3;
9      mov    dword ptr [ebp - 20], 3
10     # a.y = 4;
11     mov    dword ptr [ebp - 16], 4
12     # push b as second argument
13     lea    eax, [ebp - 44]
14     push   eax
15     # push &a as first argument
16     lea    eax, [ebp - 20]
```

```
17      push   eax
18      # push retaddr2
19      call   f2
20      # spill return value (ptr) to stack
21      mov    dword ptr [ebp - 12], eax
22      # write 24 to stack as arg1 (no push necessary)
23      mov    dword ptr [esp], 24
24      # push retaddr2
25      call   f3
26      # load a.x
27      mov    ecx, dword ptr [ebp - 20]
28      # write a.x to b[eax].x
29      mov    dword ptr [ebp + 8*eax - 44], ecx
30      # load a.y
31      mov    ecx, dword ptr [ebp - 16]
32      # write a.y to b[eax].y
33      mov    dword ptr [ebp + 8*eax - 40], ecx
34      # load b[1].x
35      mov    ecx, dword ptr [ebp - 36]
36      # load spilled variable ptr
37      mov    eax, dword ptr [ebp - 12]
38      # write b[1].x to ptr->y
39      mov    dword ptr [eax + 4], ecx
40      # deallocate stack frame
41      add    esp, 16
42      # pop saved ebp
43      leave
44      # pop retaddr1
45      ret
```

Crucially, this prevents generation of precise use-define chains between reloaded values and the sites at which they are spilled. Although the write through `ptr->y` can be linked with the value returned by `f2`, the compiler also has to consider indirect writes as potential definitions. Because of this, an alias analysis, for example, cannot narrow down that access to variables `a` and `b`, which diminishes any ability to reason about the program even further. This analysis hazard affects not only pointers, but any complex expression spanning multiple instructions involving values loaded from the stack. We found this to be an issue in all binary recompilers and identified it as

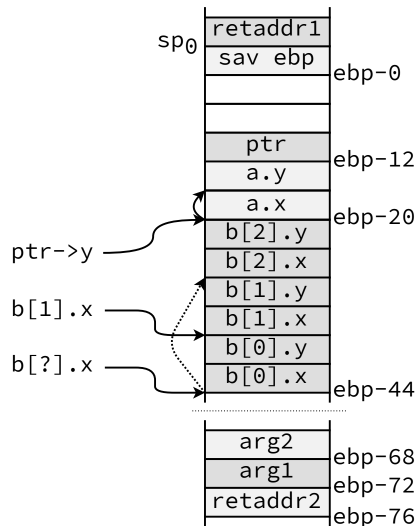

Figure 3.1: Visualization of the stack frame as allocated by the compiler corresponding to the assembly in Listing 3.2.

the primary cause limiting the efficacy of program analyses and transformations. Liu et al. have confirmed this finding in their study of binary recompilers [32].

3.2 Usage-Based Variable Symbolization

As outlined in Section 2.1.3, symbolization is the process of labeling direct references to variables with symbols that denote distinct variables. Direct references to stack-allocated variables are only found within the program text of the function owning the frame. They are indirectly encoded as a series of constant offsets relative to the initial value of the stack pointer $\texttt{sp}_0$ at the start of a function. For example, consider the array-access `b[1]` in line 19 in Listing 3.1 (line 35 in Listing 3.2). The function computes the address to this element using the value of `ebp` as base. The `ebp` register itself holds the value $\texttt{sp}_0$ `- 4`. Hence, the pointer computed by this instruction can be expressed as $\texttt{sp}_0$ `- 4 - 36`. To symbolize this access, the stack frame has to be partitioned into individual variables. Ideally, an analysis would determine that the frame contains an array $\hat{\texttt{b}}$ at an offset of $\texttt{sp}_0$ `- 4 - 44` with a size of 24 bytes. Using this information, the aforementioned reference `ebp - 36` can be labelled with an expression relative to the recovered variable $\hat{\texttt{b}}$ `+ 8`.

Despite the apparent low complexity of this function, it is remarkably difficult to identify distinct local variables. Accesses to members of composite types, such as in lines 15 and 16, are folded into direct offsets to the frame pointer in optimized binaries and do not reveal their underlying structure. In order to determine the actual bounds of variable `a`, an analysis has to establish that the access through `ptr` to `ebp-20` in line 19 can refer to the 8-byte memory area allocated for variable `a`. If this condition is met, the offset of `4` and the following write reveal `a`'s total size of `8`. At the same time, the indirect access to `b` in line 18 might access `a` or any other object in the frame, unless an analysis can provide explicit bounds for the return value of `f3`. As mentioned in Section 3.1, this is often not possible. If the bounds of the access cannot be determined, conservative static approaches are forced to label all references to local variables of a function with a single symbol.

Even if the stack frame has been perfectly partitioned into its individual variables, labeling all references in the function with the correct symbol is another challenge. In C and C++, any expression that results in a pointer that is out-of-bounds relative to its underlying array is undefined behavior, even if the pointer is not dereferenced [26]. However, that does not prohibit compilers from generating code that computes pointers lying outside the objects they refer to. For instance, compilers can turn certain index-based iterations over arrays into pointer-based iterations. Combined with other optimizations, the "end"-pointer that is used in the termination condition of such loops points lies, in rare cases, outside its corresponding array.

Listing 3.3 and Listing 3.4 illustrate one common code-pattern for which GCC consistently emits code that compares pointers outside the object over which they iterate. Two optimizations transform the code from the first into the second variant. Since the index of the outer loop is incremented monotonically by 1 in each iteration, the compiler can easily transform the outer loop to use pointers that act as iterators over the array. This allows the compiler to elide the integer variable `i`. The compiler also knows that the loop is executed at least once, but in the unoptmized variant, the condition and the associated conditional branch are evaluated 5 times. GCC exploits this knowledge to move the comparison to the end of the loop. The way this transformation is

implemented increments the initial pointer of the outer loop such that it points to the end of the first element. This causes the termination condition to compare the pointer `i` with an address that is 64 integers past the end of the array. In fact, when compiling the code in Listing 3.3, the comparison pointer lies outside the stack-frame of the function. If the body of the loop is small, as in this example, the lifter might be able to normalize the pointers and correctly symbolize pointers to the array. However, if the body of the loop is large, the values of the incremented pointer *and* the comparison pointer might be spilled to the stack, and the association between the instruction that computes the comparison pointer and the array it belongs to might be difficult to infer. In other words, a computed pointer to the stack that is subsequently written to memory cannot be automatically associated with variables that are allocated in the same position. Instead, the lifter has to consider how it is used after reloading it from the stack in order to identify the object the pointer is associated with.

Listing 3.3: Accessing a multidimensional array in a nested loop before optimization.

```
1  void out_of_bounds_ptr()
2  {
3      int stack_array[4][64];
4      init(stack_array);
5
6      // This conditional jump is executed **5** times.
7      for (int i = 0; i < 4; i++) {
8          int* arrptr = stack_array[i];
9          int accu = 0;
10         for (int j = 0; j < 64; j++) {
11             accu += arrptr[j];
12         }
13         use(accu);
14     }
15 }
```

Listing 3.4: Accessing a multidimensional array in a nested loop after optimization with GCC.

```
1  void out_of_bounds_ptr()
2  {
3      int stack_array[4][64];
4      init(stack_array);
```

```
5	    int *i = &stack_array[1];
6	
7	    do {
8	        int *j = i - 64;
9	        int accu = 0;
10	        do {
11	            accu += *j;
12	            j += 1;
13	        } while (j != i);
14	        use(accu);
15	        i += 64;
16	    } while (i != &stack_array[5]);
17	    // Notice the comparison with the address that is
18	    // 64 integers past the end of the array.
19	    // This conditional jump is executed **4** times.
20	}
```

3.3 Overview of Wytiwyg

As seen in the previous section, identification and symbolization of variables is a data-flow problem. The lifter has to consider how a stack-reference is used throughout the program to determine which variable it refers to, and how large that variable is. Since this requires high precision and detailed knowledge of the program's dataflow, we propose a dynamic and iterative approach, "Wytiwyg", to symbolize local variables in lifted binaries.

Figure 3.2 illustrates Wytiwyg's binary recompilation process in two phases. First, we rely on BinRec [3] to recover the target binary's CFG. We chose BinRec because it is the only lifter that can reliably recompile SPECint 2006 programs. BinRec uses a binary tracer (S2E [16]) that records all control transfers of the program with a user-provided set of inputs. Based on the CFG, a machine code to LLVM translator lifts the binary to LLVM IR using the instruction emulation approach outlined in Section 2.2.1. This program can already be recompiled, but it lacks variable information.

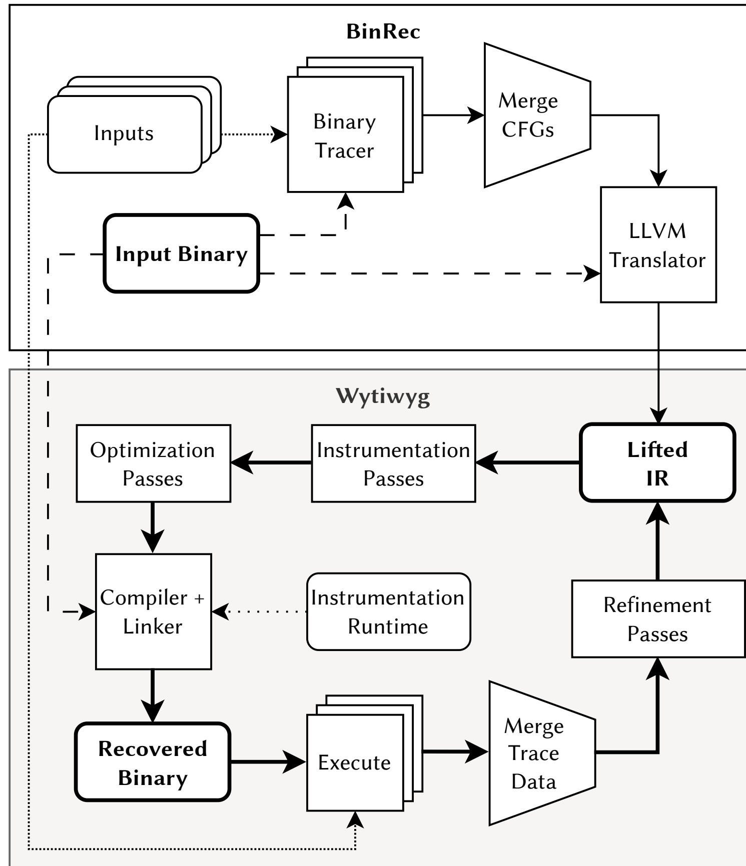

Figure 3.2: Overview of Wytiwyg. The upper section corresponds to the original BinRec recompiler. The lower section outlines our contribution. The bold transitions correspond to the *Refinement Lifting* process.

Our main contribution comprises the symbolization phase, which is outlined in the lower section of Figure 3.2. We leverage the existing capability to lift, modify and recompile programs to instrument and rewrite the lifted programs in an iterative process. Specifically, we split the symbolization process into multiple steps with dedicated dynamic analyses and IR-level transformations. Because the lifted program can be transformed like any other LLVM program, we can easily instrument the lifted IR and implement all our dynamic analyses in an external library (similar to sanitizer implementations in LLVM). At the end of each iteration, the program is transformed according to the analysis results, such that they are immediately available in the next iteration. Splitting up the symbolization process into multiple steps with dedicated dynamic analyses drastically reduces the complexity of the required analyses. This is similar to how certain compiler transformations

are used to simplify and canonicalize the IR in regular compiler pipelines. Because this process refines the lifted program's IR and improves its quality in every iteration, we call this approach *Refinement Lifting*.

3.4 Dynamic Stack Symbolization

To symbolize local variables, Wytiwyg employs two refinements. The first refinement identifies all direct stack references and rewrites them into expressions relative to sp_0. The second then determines the maximal offsets of pointers derived from each direct stack reference, uses this information to compute a stack layout for each function, and labels all direct references with symbols referring to variables within the stack layout.

3.4.1 Stack Reference Identification

To symbolize local variables comprehensively, we first need to identify all values throughout the program that constitute a direct reference to the program's stack memory. As explained in Section 3.2, direct stack references are pointers within a function that are computed as a sequence of constant displacements to sp_0. By folding all uses of sp_0, we can identify all direct stack references, and simplify them by replacing them with expressions of the form sp_0`+offset`. Consider the `push` instruction corresponding to `arg1` of the call to `f2` in line 17 in Listing 3.2. The instruction emulation initially turns this instruction into this pseudo-IR:

```
@vcpu.esp = @vcpu.esp - 4;
*@vcpu.esp = @vcpu.ebp - 20;
```

After identifying all displacements, these instructions are replaced with the following expression:

$$\texttt{*(\%sp}_0\texttt{ - }\overbrace{\texttt{4}}^{\textit{push ebp}}\texttt{ - }\underbrace{\texttt{64}}_{\textit{sub esp, 64}}\texttt{ - }\overbrace{\texttt{4}}^{\textit{push eax \# arg2}}\texttt{ - }\underbrace{\texttt{4}}_{\textit{push eax \# arg1}}\texttt{) = \%sp}_0\texttt{ - 28;}$$

However, not all uses of sp_0 can be trivially simplified, since registers holding intermediate stack references are frequently spilled onto the stack in function prologues and epilogues. In our example, `f1` saves and restores the `ebp` register during the first `push` and the `leave` instructions. From the recompiler's perspective, it is not apparent that the value of `ebp` is preserved across the invocation of `f1`. Instead, `ebp` appears to be assigned an opaque value loaded from memory before returning from the call. If `ebp` holds an intermediary stack reference (e.g., the frame pointer of the calling function) before the call, none of the pointers derived from it after the call can be folded into an offset of sp_0.

To address this, we determine for each register used in a function, whether it is merely saved on the stack for the duration of the call, or whether it is part of the function's signature. Unfortunately, indirect accesses, as for example the write to `b` in line 18, could modify any value stored on the stack and therefore complicate determining whether `ebp` is a saved register (refer to Section 3.2).

Saved registers are often identified through heuristics, that rely on platform ABI conventions to codify, which registers are to be saved to the stack before they can be used in a function, and which registers are used to transfer arguments and return between the caller and the callee (such as the System V ABI [37]). However, compilers (and sometimes developers) can disregard these conventions for functions that are not exported to other translation-units and define their own conventions on a per-function basis. Additionally, if function recovery cannot be performed with perfect accuracy, registers might not be saved and restored at the start and end of the function, and they can be saved multiple times. This can happen when tail-called functions are merged into their caller (refer to Section 3.5.1). For these reasons, identifying stack references based on heuristics is not reliable.

To avoid these issues, we use a dynamic analysis instead. Upon function entry, we assign each register a symbolic value and track how this value is used throughout the function. We consider a register saved, if the following conditions are met:

- Its symbolic value is only used in load- and store-operations from and to the current function's stack frame. If the symbol is written to any other location or used in any operation, we treat the register as an argument to the function.
- When the function returns, the virtual register contains its initially assigned symbol.

Sometimes, a register used to pass an argument is not explicitly used throughout the called function's entire body, but is "forwarded" to another function. For example, in Listing 3.2, the register `edx` is not used once. Assuming that `f2` uses the register `edx`, without knowledge of function signatures, `f2` could either save `edx` to the stack, or use it as an argument. If `edx` is used as an argument within `f2`, we need to ensure that it is passed as an argument to `f1` as well. In a situation like this, we examine a register's usage within the function it is forwarded to in order to determine whether it is saved. By default, we consider each forwarded register as saved, unless the aforementioned conditions are violated by the function the register is forwarded to. If they are violated, we know the register is an argument to the forwarding function.

Since registers can be forwarded through multiple functions until they are used, we defer evaluating the state of forwarded registers until tracing is complete. During tracing, we only record whenever a register symbol is forwarded to another function. Afterwards, we use this information to generate constraints for each forwarded register. In our example, we would produce the constraint "if `edx` is used as an argument in `f2`, then it is also an argument to `f1`". If that constraint is fulfilled, `edx` will be explicitly marked as an argument to `f1`.

Having identified saved registers for all functions in the binary, we preemptively save and restore these registers at all call sites:

```
%tmp_ebp = @vcpu.ebp
call f1   # saves ebp
@vcpu.ebp = %tmp_ebp
```

This transforms the indirect dependency on the value of `ebp` saved to and restored from the stack within `f1` into a direct dependency on the register's value `%tmp_ebp` from before the call. This IR refinement therefore substantially simplifies the identification of stack references through register spills. After folding all constant offset to sp_0, all direct references to objects on the emulated stack are expressed in terms of sp_0. These rewritten references serve as "base pointers" to local variables in the next refinement.

3.4.2 Object Bounds Recovery

Having identified all direct stack references, this refinement's purpose is to determine the layout of each stack frame and assign stack references to the identified variables. Wytiwyg uses a bottom-up approach to divide a stack-frame into distinct variables. At this point, it is unknown which references refer to the same object. Hence, we initially consider each stack reference provided by the previous refinement as a base pointer to a distinct local variable. Then, we use a dynamic analysis to record the relative minimum and maximum offsets of pointers derived from each base pointer. This yields an interval for each base pointer that indicates the underlying object's size. Expressing these intervals as ranges in terms of sp_0 yields continuous sections within each stack frame that belong to the same variable.

To generate the stack layout, we merge all ranges that are overlapping with each other and assign their associated base pointers the same symbol. For example, consider the references `ebp-44` and `ebp-36` to variable `b` in lines 29 and 35 of Listing 3.2. Initially, we assume that these two pointers belong to different objects. Once the dynamic analysis observes an access to the third element of

the array, the former pointer's interval will be recorded as `[0;20]` (offset of `16` and access size of `4`). Since this subsumes the latter pointer's interval `[0;4]`, they belong to the same object.

This also means that if `f3` from our example returns `0` in every invocation across all traces, the array will be split into two distinct symbols. Since the generated layouts are the product of actual executions, this approach ensures, for the provided set of inputs, that all base pointers are associated with the correct symbol and that the symbolized variables are sufficiently large without causing unpredictable out-of-bounds accesses (see also Section 6.2).

Tracing Runtime Overview

To track direct and indirect stack references, we employ a runtime, which is illustrated in Figure 3.3. We associate every previously identified base pointer with a unique `id`. For every `id`, we allocate a `StackVar` within our runtime, which records the bounds of the corresponding base pointer. We do not track the address of the associated base pointer in its `StackVar`, because one `StackVar` can be associated with the same variable in multiple stack-frames of the same function in recursive call-chains. As the program executes and derives new pointers from existing ones, the instrumented binary informs the runtime to update the bounds of individual `StackVars`.

To track whether an LLVM-value refers to a `StackVar` during execution, we associate each LLVM-instruction with a `PointerInfo`. Apart from a pointer to the currently referenced `StackVar`, this metadata also records the offset from the variable's base pointer. We allocate these `PointerInfo` objects for each function on the native stack, because a single logical LLVM-value can point to multiple different objects in recursive calls to the same function. Since x86 does not distinguish between pointers and integers, it is not always possible to determine statically whether a value loaded from memory is a pointer. Hence, we track this metadata for every pointer-sized integer. Additionally, we maintain a mapping of memory-addresses to `PointerInfo` which is updated whenever a pointer to a stack variable is written to or read from memory.

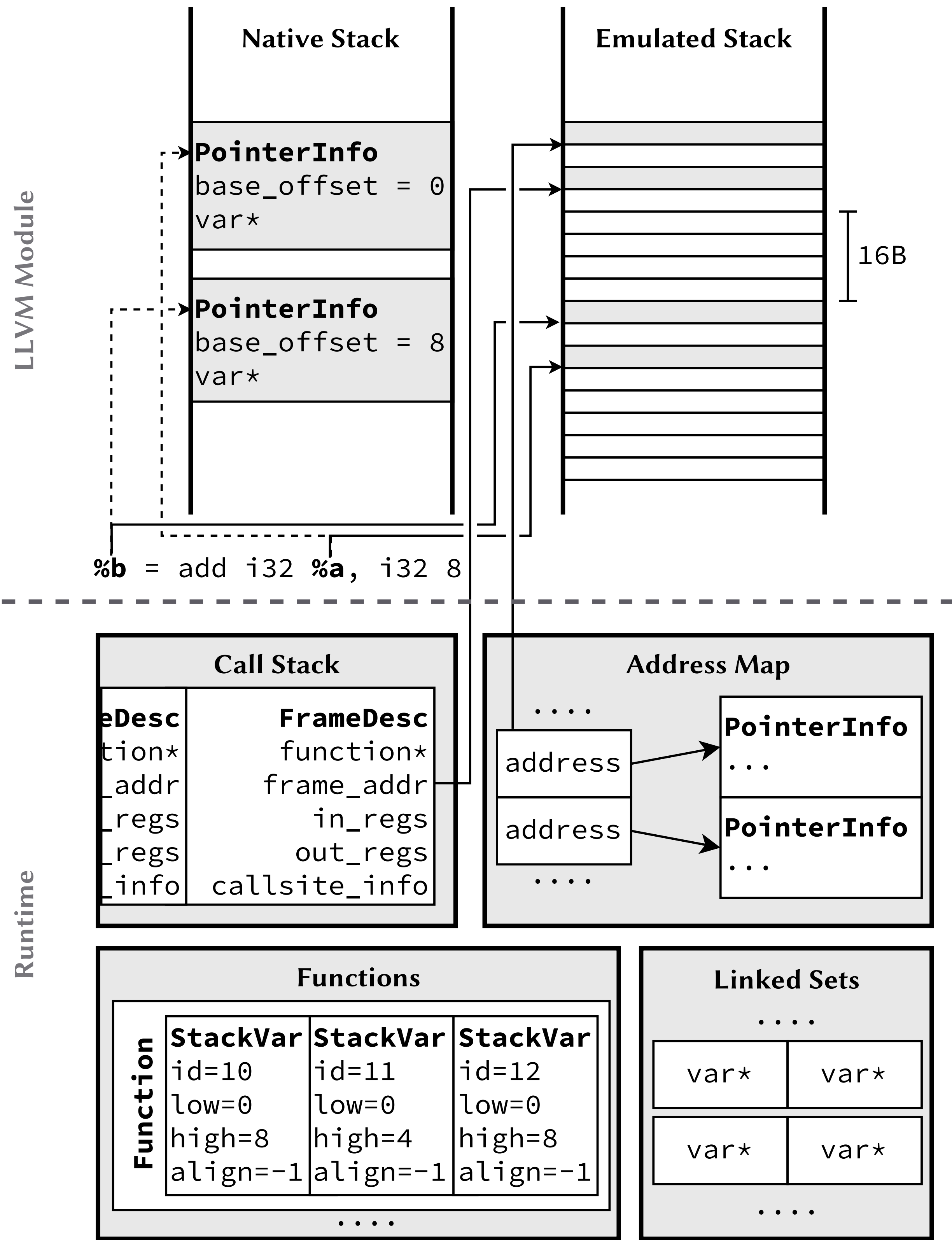

Figure 3.3: Overview of our tracing runtime.

Core Tracing Operations

In the following, we explain the central runtime operations that we instrument lifted programs with. The arguments to all operations refer to tracked LLVM instructions, which are pairs of their concrete values (i.e., the result of the instruction) and their associated `PointerInfo` metadata. The operations we implement to track pointers are:

- **`derive`** `(derived, base)` indicates that `derived` is a value derived from `base`.
- **`derive2`** `(derived, lhs, rhs)` is similar to **derive**, but is used when `derived` refers to a binary operator for which both operands could be pointers to stack variables.
- **`link`** `(a, b)` marks that the operands `a` and `b` belong to the same stack variable.
- **`store`** `(value, pointer)` records that any stack variable reference contained by `value` is written to the address specified by `pointer`.
- **`load`** `(value, pointer)` indicates that `value` contains any stack variable reference previously written to the address specified by `pointer`.
- **`copy`** `(dst, src)` assigns `dst` the pointer info of `src`.

We use **`derive`** to instrument pointer-sized `add`, `sub` and `and` instructions that have one constant operand. If `base` is associated with a `StackVar`, we initialize the metadata of `derived` accordingly. Additionally, for `and` instructions, we capture the alignment factor in the associated `StackVar`.

If an `add` or `sub` instruction does not have a constant operand, we instrument it with **`derive2`**. If exactly one of its operands points to a stack variable at runtime, we forward the arguments to **`derive`** with the known pointer operand as `base`. If both operands of a `sub` instruction are pointers, the result is the difference between the two pointers. Here, we call **`link`** instead to record

that both pointers belong to the same variable. The same applies to `cmp` instructions. Linked variables are stored by the runtime as pairs in a hash set.

The **`load`** and **`store`** operations are inserted for the LLVM-instructions of the same name. They record any pointers to stack variables that are written to memory by updating the *Address Map*. Additionally, if we load or store from or to a stack variable, we update the upper bound of the `StackVar` associated with the pointer by the size of the memory access (e.g., a store of a 32-bit integer to a pointer will update its upper bound to `4`).

To simplify the program, we turn virtual CPU registers into SSA-values before instrumentation. Hence, we need one last core operation **`copy`** to support PHI-nodes. This operation is used to copy the `PointerInfo` of incoming values to the PHI-node's associated `PointerInfo`. We insert these at the predecessor-blocks for each `phi` instruction.

False Derives

Deriving a value from a pointer does not always yield a new pointer. Writes to 16-bit or 8-bit x86 sub-registers do not zero out their upper 16 or 24 bits, respectively. This is prevalent in C++ code using booleans. Because there can be arbitrarily complex code between writing to and reading from such a register (including crossing function boundaries), it might not be possible to determine that the upper bytes of a register are stale. This creates a false dependency on the value previously stored in the register. This is problematic if the previous value is a tracked pointer, because the result of a subregister store appears to be a pointer, but might be an entirely unrelated value. The only way to confirm the value's validity as a pointer is to wait until it is dereferenced. For this reason, we do not update the bounds of stack variables in the **`derive`** operation, but only within the **`load`** and **`store`** operations.

Out-of-Bound Pointers

Deferring updates to the bounds of stack variables until a pointer is dereferenced addresses out-of-bound pointers only partially. It correctly handles the case exemplified in Listing 3.4. However, sometimes, the base pointer itself can be out-of-bounds of the variable it refers to. In that case, we need to defer *initialization* of the bounds until the first access, instead of automatically assuming that the base pointer is part of the object. Hence, we update bounds according to the following conditions:

- The bounds of `StackVars` are initially undefined.
- The first time any pointer associated with a `StackVar` is dereferenced, the lower *and* upper bounds are initialized with that pointer's offset.
- When linking two `StackVars`, their ranges are only merged if both have defined bounds.

Function Calls

Arguments to functions are usually pushed to the stack by the caller. Hence, we not only observe direct accesses to the stack frame of the current function, but also to the frame of the function invoking it. Because functions can have a variable number of arguments, we record such accesses per call-site.

To facilitate this, the runtime keeps track of the call stack and records, for each stack frame, its base pointer (i.e., sp_0) and the call site through which the function was entered. A call site descriptor comprises, similar to a function, a list of stack variables, a `PointerInfo` referring to the stack variable containing the argument list in the original frame, and an `id` that lets us map it to its corresponding LLVM call-instruction. The **`derive`** operation treats all base pointers with an offset

greater or equal to the current frame's sp_0 as arguments. These accesses are recorded in the call site descriptor of the currently active frame.

To keep track of the current call-stack in our runtime, we instrument every LLVM call instruction with the following operations:

- **`fnenter`** `(function*, callsite, regs)` creates a new frame descriptor for the specified `function` that is associated with the specified `callsite`.
- **`fnexit`** `(regs)` pops the current stack frame from the runtime call stack.

Both operations additionally have a list of registers as arguments to marshal metadata associated with virtual registers between calls.

Replacing Base Pointers

Tracing yields bounds and alignments for base pointers, argument lists for call sites, and a list of linked base pointers. After coalescing overlapping and linked base pointers, we generate function signatures. First, we merge all call site signatures for a function into a *super signature*. We propagate the super signature back to all call sites to fill in gaps in argument lists. For example, consider a call site for a function with four arguments. Suppose we only traced accesses to the first and third argument. Here, we would fill in the second argument. We omit the fourth argument, because it could be a directly accessed variable argument. However, if the function's variable arguments are accessed indirectly, the variable arguments will be passed as a pointer to an array allocated in the caller's stack frame.

Once the signatures of all call sites have been determined, we add function arguments that were passed on the stack to the lifted function's signatures. Then, for each coalesced base-pointer set, we allocate variables with the deduced sizes and alignments. Finally, we replace all base pointers

with pointers to the newly allocated variables. We ensure we preserve the alignment of these new pointers if they refer to objects for which we observed they are used in alignment operations. At this point, we can remove the emulated stack from the lifted binary, since all dependencies on the emulated stack were replaced with native stack allocations.

3.5 Implementation

We implemented Wytiwyg as an extension to BinRec, because it is, to our knowledge, the only dynamic binary to LLVM IR lifter and recompiler capable of translating COTS binaries reliably. We upgraded the LLVM version used by BinRec from 3.8 to 14 and rebased the S2E plugins used for exporting traces onto upstream S2E [16]. Rather than translating the machine code to LLVM IR while the program is running, we use a modified version of RevGen [15] to lift the program offline after completion of the initial tracing. We found this to accelerate the initial tracing drastically and eliminate complexity originating from merging LLVM modules containing the unprocessed traces. Finally, we incorporated a driver that executes tracing, translation and application of refinements automatically.

At the time of development, BinRec did not support lifting of x86 64-bit binaries. Therefore, our prototype targets only x86 32-bit binaries. This does not affect the generality of our approach, because there are no fundamental differences between these two architectures in terms of how the generated code interacts with stack memory.

3.5.1 Function Recovery

The original version of BinRec did not recover functions and merged all basic blocks into one large function. Naturally, this is an unsuitable representation to lift variables that are local to

their function. Hence, we implemented a function recovery similar to the approach detailed in Nucleus [8].

Initially, we create an inter-procedural control-flow graph of the entire binary based on the control transfers that were logged during tracing. Then, we mark any block that is the target of either a direct or indirect call instruction as a function entry. Before function bodies can be computed accurately, all tail-calls have to be identified. Tail-calls are jump-instructions inserted by compilers in place of regular call-instructions. This can happen if a function `f1` calls another function `f2` with the same signature and the call to `f2` would be the very last instruction of `f1` before it would return. Like regular calls, tail-calls can be indirect and/or have a variable number of arguments. The majority of tail-calls can be identified by checking for each direct or indirect jump, whether any of its targets match the entry address of an already identified function.

Sometimes a function has no regular callers and is only encountered as a target of tail calls. Nucleus would merge such functions with their callers and classify the result as a function with multiple entries. Because LLVM IR lacks a natural representation for functions with multiple entries, our implementation splits functions such that there are no overlaps and have only one entry. Our algorithm for this simple: we compute for each function entry the set of blocks reachable through jumps. Then we count in how many functions each block is contained. If a block is contained in more functions than any of its predecessors, it is marked to be a function entry. We found this approach to work reliably across all our inputs, including ones that contain nested and/or indirect tail calls. Functions that are exclusively reachable through a single tail-call and have no regular call sites throughout the entire program are merged with their caller, however. We verified our results by cross-referencing all detected functions with the binary's symbol table (if available) and did not encounter any false positives.

3.5.2 Variable Argument Library Calls

Since arguments of calls to external functions are passed on the stack, recovering the operands of these calls is a prerequisite to full stack symbolization. If their prototypes are known, BinRec lifts such calls by loading the corresponding arguments from the emulated stack and specifying them as operands to an LLVM call instruction. Fortunately, identifying the arguments of external functions is trivial for most system library functions, because their signatures are known. However, functions that have prototypes with a variable number of arguments, such as `open` or the `printf`-family of functions, require special handling.

To lift calls to these functions, BinRec uses a mechanism called *stack switching*. Because the lifted program pushes all arguments on the emulated stack as required by the external function, the lifted program instructs the *native* stack pointer to point to the emulated stack for the duration of the external call. However, this approach is not compatible with stack symbolization. During symbolization, Wytiwyg eliminates the emulated stack, so it is no longer possible to perform stack switching. Hence, arguments to these calls have to be recovered *before* Wytiwyg can proceed with symbolization.

There is no uniform way to determine the number of arguments at call sites for variable argument functions. The full prototypes for individual call sites to such functions can usually be determined by inspecting the values of the functions' named arguments at runtime. Therefore, Wytiwyg uses an additional *refinement* before stack symbolization to fully lift calls to these functions. For example, this refinement inspects the format string passed to `printf`-style functions at runtime to determine an exact signature for each call site.

3.5.3 External Functions

Wytiwyg has to account for any effects on pointers passed to external functions. Because dynamically linked functions are not lifted, we can only instrument calls to them. In our implementation, we maintain a database of known external library functions together with their signatures. The majority of effects necessary for tracking pointers can be expressed through a small set of constraints on the functions' arguments and return values:

- **`ObjectSize`** `(ptr, size, count)`: The object specified by `ptr` is at least as large as the product of `size` and `count` (e.g., `fread`).
- **`ZeroTerminated`** `(ptr)`: The data that `ptr` points to is zero-terminated (such as C strings).
- **`Derive`** `(derived, base)`: The pointer `derived` refers to the same object as the pointer `base` (e.g., `strtok`).
- **`Clear`** `(ptr, [size])`: The external function will clear out any references to stack variables stored in the object that `ptr` refers to (such as structure initialization or `memset`).
- **`Copy`** `(dst, src, [size])`: The external function will copy any references to stack variables stored in the object `src` to the object `dst` (such as structure initialization or `memset`).
- **`FormatStr`** `(str, valist)`: The argument `str` is a C-style format string that describes the arguments contained in a standard C `va_list` (e.g., `vprintf`).

During instrumentation, we translate these constraints into the tracing operations documented in Section 3.4.2.

3.5.4 Non-Deterministic Code

For certain programs, the same input can cause different paths across multiple executions. There are many reasons for this to happen. The execution could depend on timing, randomness, pointer-values, or other factors that cannot be easily controlled. If a path was not driven during execution, then there is no analysis data available to symbolize variable references on it. BinRec/Polynima (refer to Chapter 4) is already capable of detecting control-flow misses (i.e., a direct or indirect jump to an untranslated address) in the recompiled binary. If such a miss is detected, the offending address is merged incrementally with the inter-procedural control-flow graph and the binary is re-recompiled. However, variable references on paths that could not be driven during tracing would still refer to the emulated stack.

Because our current implementation depends on instrumenting the whole binary, we cannot selectively trace a subset of blocks. For our target benchmarks (*Xalan-C++* and *GCC*), we found this to be an issue, in particular with hash sets and maps that rely on pointer-values as keys. Because of the varying key-values, different paths checking for collisions would be exercised across different runs with the same input. Without the ability to drive execution through these collisions selectively, we cannot symbolize base pointers in these paths correctly (since our tracing data contains no information on these base pointers). To handle this, we statically coalesce these pointers with the inferred stack layout. This risks incorrect symbolization because of potential out-of-bounds pointers (as described in Section 3.2). However, we found that not to be an issue for our target benchmarks, since these difficult-to-trace paths are usually not occurring together with the problematic iteration pattern.

3.5.5 Untangling Large Stack Frames

Functions with very stack frames spanning multiple (4 KB) pages require some extra work to set up the frame in their prologue. On Linux, rather than allocating a fix amount of space for

a program's stack, the kernel grows the stack as needed. This is achieved by reserving a guard page at the end of the physically allocated stack. When the program tries to access this guard page, a page fault is raised, and the kernel allocates a new page for the stack. When a function's stack frame spans multiple pages, it might move the stack pointer past the guard page into an unmapped region. On accessing this unmapped region, the running program would crash with a segmentation fault. To avoid this, functions with large frames have to ensure in their prologue that all pages are allocated before they are used. This is usually achieved by decrementing the stack pointer and performing a memory access in a loop one page at a time until the frame has reached the desired size (compare with Listing 3.5).

Listing 3.5: Setting up large stack frame on page at a time.

```
1  func():
2      # ...
3      lea     r11, [rsp-0x7000]  # Target address
4  .L1:
5      sub     rsp, 0x1000        # Decrement by one page
6      or      qword [rsp], 0x0   # Access the page to allocate it
7      cmp     rsp, r11           # Compare with target address
8      jne     .L1                # Loop until target address is reached
9  .L2:
10     # ....
```

Our analysis is based on the assumption that variable base pointers are expressed as immediate offsets from sp_0. If the stack pointer is decremented in a loop, the base pointers of the variables in the stack frame are expressed in terms of the loop. This would turn the stack into one large variable. To avoid this, we have to untangle this loop and replace all uses of the stack pointer after the loop with the final value of the stack pointer. Since this loop monotonically increments the stack pointer, we can use LLVM's Induction Variable Simplification that automatically computes the value of the stack pointer and replaces all uses of the stack pointer that depend on the value computed by the loop.

3.6 Evaluation

We evaluate our implementation in three ways. First, we verify the ability of Wytiwyg to symbolize programs from the SPECint 2006 benchmark suite while retaining their functionality. Each target benchmark is compiled in multiple configurations with different compilers and optimization levels to ensure a broad range of inputs. Second, we estimate the "quality" of the generated IR. Since there is no agreed-upon benchmark to measure IR-quality, we rely on performance measurements of the recompiled binaries as a proxy indicator for IR-quality. Language frontends strive to produce IR that is best understood by passes that are part of LLVM, and developers within the LLVM ecosystem optimize their downstream applications to process IR emitted by those frontends. Since LLVM's primary purpose is program optimization and efficient code generation, we argue that a decrease in runtime overhead indicates that the refined programs better match the expectations of LLVM and LLVM-based tools. Last, we conduct a comparative analysis of the inferred stack layouts against the ground-truth data provided by LLVM to ascertain the accuracy of our approach.

We target the SPECint 2006 benchmark suite, which has been widely used in previous binary lifting and recompilation literature [3], [6], [22], [32]. This benchmark-suite comprises single-threaded real-world programs, which makes them an ideal target to evaluate the impact binary recompilers have on performance and correctness. We exclude the `omnetpp` and `perlbench`, because our prototype does not handle `setjmp`/`longjmp` and C++ exceptions. We do not evaluate on the SPECfp 2006 set of programs, because x87 instructions are translated using QEMU's software float emulation, and our current implementation does not convert these to LLVM floating-point instructions.

Liu et al. identified BinRec [3], McSema [48], RetDec [27] and mctoll [53] as the best available binary lifters targeting a compiler-level IR [32]. According to their paper, McSema is the only static lifter able to recompile a subset of the SPECint 2006 benchmarks. Although McSema can symbolize stack variables using IDA Pro's stack analyses, the authors admit this process is not automatic

because of the heuristic nature of IDA Pro's analyses [21]. For these reasons, we compare Wytiwyg with SecondWrite, which was provided to us by its authors. To our knowledge, it is the only binary to LLVM IR lifter that claims to be capable of recompiling most of the SPECint 2006 benchmarks and supports the symbolization of stack variables.

3.6.1 Functionality

The primary goal of our approach is to recover high-level semantics in binaries without relying on heuristics tailored to the program, compiler or optimization level. To assess whether we achieve this, we compiled each benchmark in multiple configurations. We use the latest GCC 12.2 and Clang 16 compilers at their highest optimization level `-O3`. Additionally, we compiled one set of unoptimized benchmarks with GCC 12.2. SecondWrite could disassemble none of the benchmarks because certain SIMD instructions are not handled by their translator. Hence, we also compiled all benchmarks using GCC 4.4.3 with optimizations enabled on Ubuntu 10.04. This is very close to the GCC version used in the original evaluation of SecondWrite (GCC 4.4.1). We note that, while we did not pass any flags to GCC 12.2 or Clang 16 to emit architecture-specific instructions, older versions of GCC do not emit SSE instructions by default. Since BinRec implements SIMD instructions in software using helper functions provided by QEMU, instruction compatibility is not a concern for Wytiwyg.

We used the `ref` datasets as inputs to trace and validate the recompiled binaries. Wytiwyg successfully lifts and recompiles all binaries and inputs, with no manual intervention. Because the `gcc` and `xalancbmk` benchmarks make use of hash maps using pointers as keys, different executions of the same inputs explore different paths in the lifted binary. We used BinRec's incremental lifting to generate sufficient coverage for these binaries [3]. For the same two benchmarks, we increased the maximal allowed stack-sizes (using `ulimit -s`) due to deeply nested recursive call-chains. Wytiwyg turns tail-calls into regular calls, and the LLVM-signatures

Table 3.1: Normalized runtime of recompiled binaries relative to the runtime of their respective input binary for each configuration.

		BinRec / Wytiwyg				SecondWrite
no symbolize	✗	GCC 12.2		Clang 16	GCC 4.4	GCC 4.4
symbolize	✓	-O3	-O0	-O3	-O3	-O3
bzip2	✗	1.15	0.74	1.21	1.06	0.94
	✓	1.03	0.51	1.13	0.85	0.91
gcc	✗	1.39	0.82	1.58	1.18	—
	✓	1.22	0.49	1.25	0.89	—
mcf	✗	0.99	0.75	1.09	0.97	0.98
	✓	0.92	0.65	1.07	0.88	1.08
gobmk	✗	1.25	0.99	1.20	1.20	—
	✓	0.99	0.79	0.97	0.91	—
hmmer	✗	2.38	0.67	1.59	0.72	0.99
	✓	3.04	0.48	1.30	0.60	0.98
sjeng	✗	1.06	0.79	1.13	1.09	1.16
	✓	0.85	0.62	0.87	0.82	1.11
libquantum	✗	1.15	0.92	1.57	1.16	1.26
	✓	1.21	0.70	1.14	0.89	—
h264ref	✗	1.35	0.83	1.60	1.05	1.75
	✓	1.01	0.48	1.23	0.84	1.73
astar	✗	0.95	0.69	1.04	0.96	1.08
	✓	0.79	0.47	0.91	0.80	1.08
xalancbmk	✗	1.13	0.55	1.23	1.17	—
	✓	0.90	0.10	0.87	0.77	—
Geomean	✗	1.24	0.76	1.31	1.05	1.14
	✓	1.10	0.48	1.06	0.82	1.12

of the caller and caller do not always exactly match up in the recovered binary. This prevents LLVM from lowering these calls back to tail-calls.

We recompiled binaries with SecondWrite using default optimizations and disabling speculative disassembly. Without stack splitting, all binaries could be recompiled, except `xalancbmk` and `gobmk`. `xalancbmk` could not be linked and `gobmk` could not be processed by SecondWrite's disassembler. `gcc` crashed on every single `ref` input, even after disabling all of SecondWrite's heuristic optimizations and enabling speculative disassembly. `libquantum` crashed during execution if we enabled stack splitting.

We also noticed that SecondWrite cannot lift binaries that have been compiled with position independent code (PIC). It does not handle some types of relocations, that GCC 4.4 emits for position independent code. This is only a minor engineering defect, and we were able to produce a working binary for `mcf` by manually patching these relocations. A more significant issue that we encountered was limited support for jump tables. For example, the jump table in the PIC version of the function `BZ2_decompress` from the binary `bzip2` was entirely missing. Even when enabling speculative disassembly, the jump-targets were not present in the lifted LLVM IR. We assume SecondWrite cannot identify speculative control transfer targets if the references to them are not encoded as absolute addresses in the binary's data.

3.6.2 Performance

Our performance experiments were conducted on a system running Ubuntu 22.04 with an AMD Ryzen 9 3900X running at a base clock of 3.8GHz. We disabled frequency boosting, clock-frequency scaling and simultaneous multi-threading to produce consistent results. We instructed LLVM to target the *pentium4* architecture, to avoid measuring the impact of newer CPU features that are available on our target machine.

Table 3.1 contains the relative performance impact of recompilation and stack symbolization on each of our input binaries. Across almost all benchmarks, our stack symbolization approach significantly improves the runtime overhead of recompiled binaries, with the worst case average runtime for heavily optimized binaries at $1.10x$. However, binaries that were not compiled with the latest state-of-the-art compilers can experience a significant uplift in performance: programs compiled with GCC 4.4 see a $1.22x$ speedup, despite being compiled at the highest optimization level. Unoptimized binaries are more than twice as fast, with an average speedup of $2.10x$. Compared to the non-symbolized versions of those binaries, these performance improvements

confirm our hypothesis that recovery of fine-grained stack symbols is central to enhancing the IR-quality of lifted programs and allows for full-scale program reoptimization.

We also improved the non-symbolized baseline for unoptimized binaries compared to the original version of BinRec from $0.98x$ [3] to $0.76x$. This is partly due to upgrading BinRec from LLVM 3.8 to LLVM 14, but we found that performing function recovery enhances the results even further. Without function identification, calls were translated into jumps to the function's entry basic block and return instructions were turned into LLVM `switch`-instructions that determine the return target based on the return-address stored on the stack. In the resulting control-flow graphs, frequently called functions act as "chokepoints", because calls appear to return to entirely different call-sites. This optimization hazard is even more prevalent in unoptimized binaries, where small functions with many call sites are not inlined.

To understand the relationship between recompiled and native binaries, we compared all runtimes in Figure 3.4 with the baseline of native binaries generated by GCC 12.2. The performance of the recompiled binaries across all `-O3` configurations approaches the GCC 12.2 baseline, although `-O0` is slightly behind. This disparity can be attributed to Wytiwyg not yet recovering global or heap variables, because accesses to these variables are not optimized when compiling a program with `-O0`. Since we do not symbolize these, LLVM's ability to reoptimize these accesses in the lifted program is limited.

Despite symbolization enhancing performance in most cases, there are some outliers: `hmmer` and `libquantum`, when compiled with GCC 12.2 and optimization level `-O3`, experience a degradation in performance. This indicates that LLVM's optimization heuristics are not optimal when applied to the lifted programs. Especially the $2.28x$ ($3.04x$ if symbolized) slowdowns of `hmmer` contradict existing binary recompilation literature, where recompiling this benchmark often exhibits one of the greatest performance improvements across SPECint 2006 [3], [6]. However, Figure 3.4 reveals, that more recent compilers are able to drastically reduce the runtime of this benchmark, to where

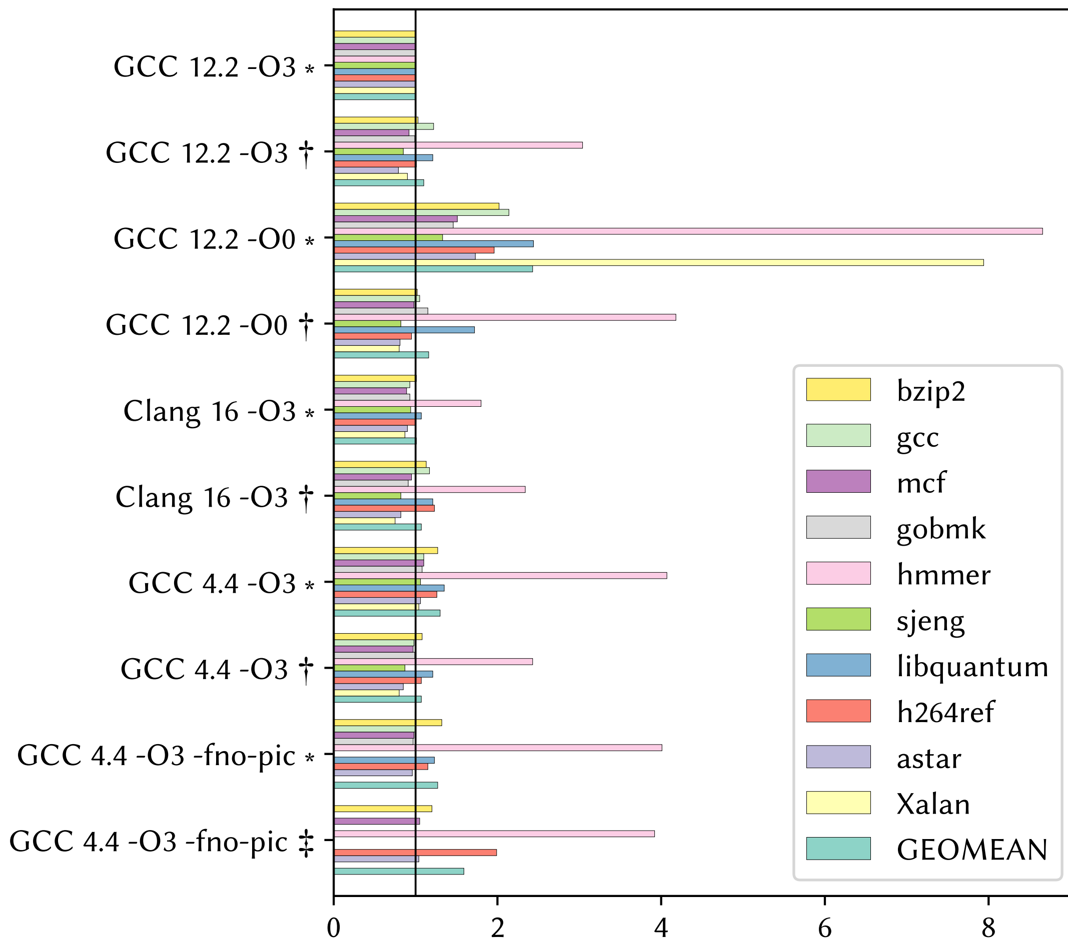

Figure 3.4: Normalized runtime of input (∗) binaries, binaries recompiled and symbolized with Wytiwyg (†), and binaries recompiled and symbolized with SecondWrite (‡) relative to the runtime of the respective binaries compiled and optimized with GCC 12.2.

a $3.04x$ slowdown relative to the GCC 12.2 binary is still faster than the binary produced by GCC 4.4.

We found that vector instructions in the original binaries can cause non-optimal code after lifting. Although LLVM often synthesizes the software-emulated SIMD instructions into LLVM intrinsic vector instructions, the generated sequences are usually more verbose and less efficient. Further, if a function accesses a vector register only partially, it creates a false dependency on the value of that register before the entry of this function. If a program uses SIMD instructions only sparsely (such as `gcc`), these false dependencies can cause vector register values to be copied across multiple

function boundaries. Hence, we believe that there is room for improvement by lifting vector instructions more effectively.

Our measurements for SecondWrite diverge with the reported results [6]. Without stack splitting, in the original evaluation, they measured speedups for `libquantum`, `h264ref` and `astar` rather than slowdowns. Similarly, we did not observe a $1.38x$ speedup for `hmmer`. We verified SecondWrite is compiling the lifted IR with optimizations enabled and could not identify a reason for this disparity. After enabling function splitting, we measured an improvement of 2 percentage points, which appears consistent with the results reported in their paper. We note SecondWrite does not lift using a separate, emulated stack, and always inlines stack frames as allocations into the lifted LLVM functions. This explains to some extent the smaller improvement compared to the binaries recompiled without stack splitting.

3.6.3 Splitting Accuracy

To evaluate the accuracy of our approach, we compare the dynamically recovered stack-allocations with the ground-truth generated by LLVM 16's *Stack Frame Layout* analysis. This analysis outputs the stack layout used by code generation for each function when compiling from source. We only consider functions that were executed in our traces, since untraced functions are not contained within the lifted binary. The results are displayed in Figure 3.5. We assign each ground-truth allocation one of four categories depending on whether it overlaps with an object in the recovered layout: *matched* on perfect match, *oversized*, *undersized* and *missed* on full, partial or no overlap, respectively. Matched and oversized allocations are sufficiently large to prevent overflows, although oversized allocations potentially prevent optimizations. Undersized and missed allocations indicate that there might exist valid inputs to the original program that will cause out-of-bounds accesses in the recompiled program. We note that spilled floating-point and XMM-values can decay into multiple smaller objects due their emulation in software. Although those objects are

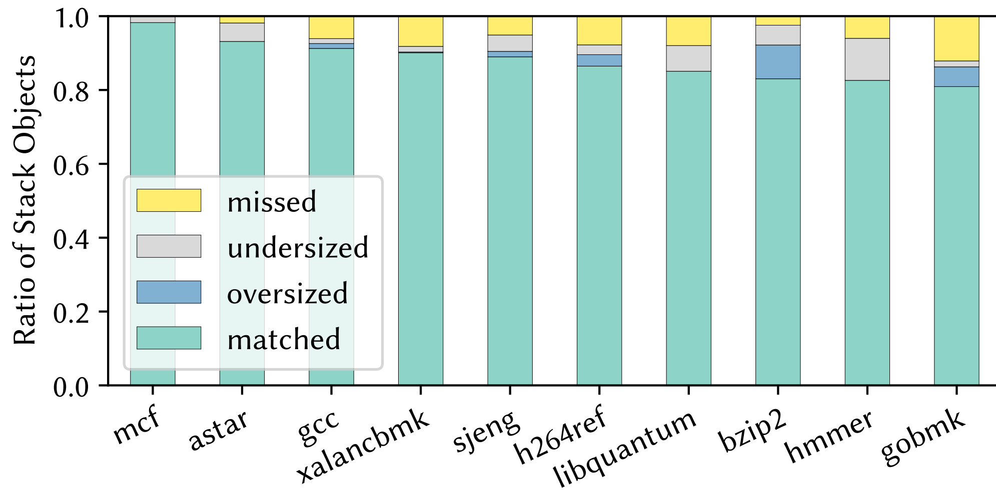

Figure 3.5: Accuracy of Wytiwyg.

safely symbolized, our evaluation classifies them as undersized. However, we believe they are only a small fraction of undersized objects. Regardless, our approach achieves a precision of 94.4% and a recall of 87.6% without providing additional inputs.

Chapter 4

Polynima—Practical Hybrid Recompilation for Multithreaded Binaries

Chapter 3 presented an iterative and incremental approach for recovering stack variables in single-threaded binaries. With some minor exceptions, the programs that were evaluated are highly deterministic and explore consistently the same control flow paths across different runs with the same inputs. This is an oversimplification of the real-world software space, as most contemporary software is multithreaded and exhibits non-deterministic behavior because of the concurrent execution of multiple threads. Crucially, *no existing recompiler* addresses the specific challenges imposed by multithreaded programs that are ubiquitous in the modern software space.

This chapter introduces Polynima, the first practical binary recompiler that supports the general lifting and recompilation of a wide range of multithreaded x86/x64 binaries. Rather than relying exclusively on static or dynamic analyses, Polynima implements a hybrid control flow recovery approach that combines the benefits of both techniques, while providing an efficient strategy for handling unknown paths.

4.1 Lifting Non-Deterministic Programs

Besides the analysis challenges outlined in Chapter 2, the non-deterministic nature of multi-threaded programs poses additional challenges for recompilation. The key promise of trace-driven recompilation techniques has been generation of recovered programs that might not contain every possible (valid) path of the original binary, but at least guarantee functional correctness for the inputs that were exercised during the trace. This guarantee guarantee cannot be extended to any multi-threaded program, because the interleaving of threads can cause execution of unexplored paths across different runs. As mentioned in the previous chapter, this issue also affects single-threaded binaries that might not even rely on explicit sources of randomness or non-determinism (e.g., `rand` or `gettimeofday`), but are subject to implicit non-deterministic conditions that vary across executions, such as hash-table lookups using pointers as keys. The recompiled programs of such binaries suffer from similar reliability issues as statically lifted programs, since they may fail randomly even when processing known inputs.

For some programs, it might be possible to limit possible sources of non-determinism artificially, such as by providing a fixed seed for the random number generator, or disable address space layout randomization (ASLR) to ensure that pointers have the same numerical values across different runs. For multi-threaded programs, this is not a viable option. The interleaving of threads is inherently non-deterministic and can lead to different execution paths, even when the same inputs are provided. Even if the scheduler is fixed, the order in which threads access concurrently shared data structures can lead to different outcomes. The only way to eliminate non-determinism in such programs is to force them to execute in a single-threaded environment. However, the associated performance penalties associated with this approach are unacceptable for most applications.

One proposed approach for handling non-deterministic programs is to lift the binary incrementally. Section 2.1.1 already provided a brief overview of the incremental lifting approach. The idea is to start with a partial recompilation of the binary that only covers the control flow paths that

are exercised during the initial trace. When during its execution a new path is discovered that is not part of recompiled binary (i.e., a "control-flow miss"), the program logs the computed target address and terminates itself. This address is then fed back into the recompiler, which executes the binary in a processor emulator/tracing engine. After conclusion of the trace, the recompiler finally merges the resulting trace into the already lifted program and generates a new recompiled binary.

This approach was first implemented in BinRec [3]. However, the proposed incremental lifting approach has only limited utility when working with non-deterministic programs. Since driving the program from its entry-point to the control-flow miss might not be trivially possible, the instruction-pointer is initialized instead with the logged address. No other state is retained from the terminated process (i.e., heap, stack, and registers remain completely uninitialized). When running the program from an arbitrary point with no initialization, it will usually crash before it reaches the next branch or call-instruction. This means that at most one linear instruction sequence can be lifted after each control-flow miss. After recompiling the program with the newly added path, execution has to be restarted from the beginning, and navigate the same path that initially triggered the control-flow miss to reveal the next address executed within that path. If that path is only taken by chance, it might take a long time until the next control-flow miss of the missing path is discovered. Crucially, the incremental lifting approach does not produce recompiled binaries that can be run without the danger of encountering control-flow misses, because it fails to generate comprehensive coverage for rarely executed paths that are part of the original program.

4.2 Design and Implementation

To address the challenges posed by non-deterministic programs, we propose Polynima, a practical hybrid recompiler that supports the general lifting and recompilation of a wide range of multithreaded x86/x64 binaries. Polynima is an end-to-end recompiler comprising modules that

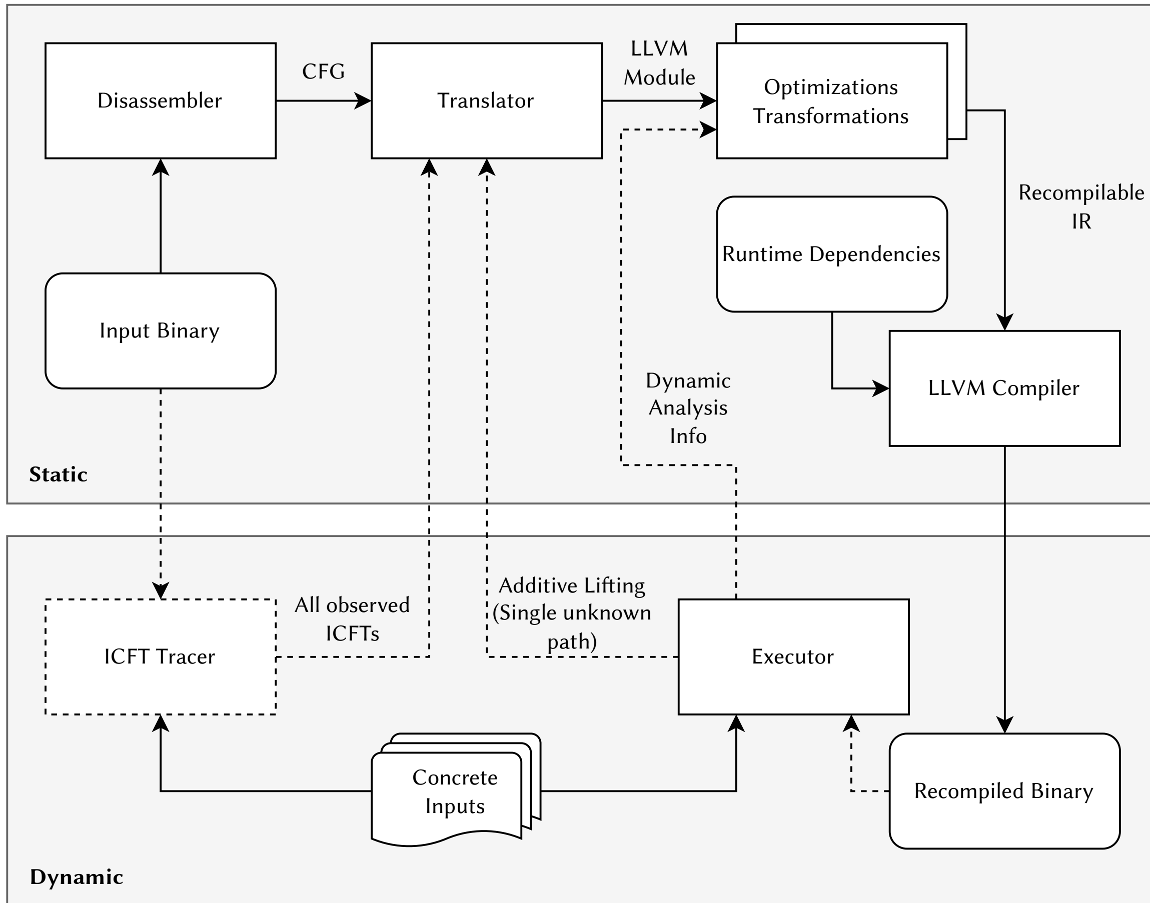

Figure 4.1: Overview of Polynima. Dashed lines indicate optional steps.

perform control flow recovery, translation of machine code to LLVM IR, optimization and lowering. Figure 4.1 summarizes the system's architecture. Recompiled programs generated through static-only analyses acts as a functional baseline replacement for the input binary. Although this initial output representation only supports control flows that are recovered through the COTS disassembler, we instrument the lifted IR to handle unknown transfers at runtime. Our optional dynamic analyses, such as those for optimizing the lifted IR, build on top of this representation.

4.2.1 Compatibility

Our prototype supports a wide range of binaries, but we impose certain reasonable restrictions on the input for implementation reasons. We support the recompilation of x86/x64 Linux-based

C and C++ binaries for their original architectures. Binaries may use threading models and synchronization primitives as exposed by POSIX threads (`pthreads`), C11 (`thread.h`), C++11 (`std::thread`) and OpenMP programming models. Supported programs may also implement custom primitives provided by functions from any of the above interfaces, such as C11 (`atomic.h`) or C++11 (`std::atomic`). We also handle compiler built-ins, such as the `__sync_*` variants, that are typically compiled using hardware atomic instructions.

Linux threads, in this case defined as lightweight processes, are spawned by calling the `clone` system-call, which is wrapped by library functions such as `pthread_create` or `thrd_create`. Polynima supports external library calls with unknown interfaces through *stack-switching*, where the native stack pointer points to the emulated stack for the duration of the call. However, if the call enters a new thread context, it would work with its own thread-local stack. Implementing stack-switching in such a scenario would involve dealing with four stacks of execution, making the implementation overly complex. For that reason, we require the knowledge of signatures for library functions that spawn threads, such that we can lift them to execute on the native stack. We do not support user-level threads that can be achieved through `get/setcontext`, `make/swapcontext`, and `longjmp/setjmp`.

4.2.2 Control Flow Recovery

For the initial lift, Polynima consumes information about function entry points, the basic blocks belonging to them, and the direct control transfers between identified basic blocks from a COTS disassembler. We treat jump and call instructions as basic block terminators and explicitly label control transfers as *jump*-based or *call*-based in the CFG. Basic blocks are labeled as *direct* if the terminator instruction encodes the transfer's target address, and *indirect* otherwise.

For indirect control transfers, we assume a set of known targets and lift them as switch statements that select their target based on the current value of the emulated program counter (PC). Each

switch case represents a value that the PC could assume in the original program, and is mapped to the corresponding lifted block in the IR. But, obtaining a set of targets for an indirect control transfer is a hard problem. Polynima thus implements a hybrid approach that can use static and dynamic analysis results. We currently support three distinct ways to achieve this.

Static

Modern disassemblers implement various heuristics to resolve jump tables and infer targets of indirect calls. Polynima consumes (*but does not require*) disassembler-provided targets for indirect jumps and calls, benefitting from advances in static CFG recovery. As the control flow is conditioned on the actual PC value at runtime, Polynima can also graciously handle incorrectly predicted targets. However, as statically collected information can be imprecise, we may observe previously unknown control flows during the execution of the recompiled binary.

Additive

To support dynamically discovered targets, Polynima implements additive lifting. We achieve this by redirecting the default case of all switch statements for indirect control-transfer (which is executed if the target address has not been observed for this branch) with a runtime function. When encountering a new path, the runtime updates the on-disk representation of the CFG with this information and then stops program execution.

Starting at this target, we perform a static recursive descent style exploration of the original binary control flow and integrate back all the discovered paths into the known CFG. This technique is useful for jump-table style control transfers, where the paths from the newly discovered block eventually join with the rest of the known CFG through direct transfers. We then rerun the recompilation pipeline to generate a new binary that supports the additional paths. The entire process can be thought of as a recompilation *loop*, with each intermediate output supporting

statically known and dynamically discovered control flow. Discovering new paths by natively executing the recompiled output is an efficient and complementary strategy for static CFG recovery techniques.

Crucially, additive lifting enables *on-device* lifting. This can be useful for recompiling legacy binary programs without access to their original execution environment or a suitable emulator. Users first statically generate a fully functional recompiled binary that lacks information about indirect control-transfers. Then, the binary is ran on the new architecture, and the recompilation process is triggered when a new path is discovered. This is possible as the recompilation process enables the linking of new libraries, patching unsupported instructions and compiling for a different ISA.

Dynamic

We note that the performance of the above approach is directly proportional to the total time required for each recompilation run. This can be inefficient when, (1) the time required for an individual lift-and-lower step is high, such as in the case of large binaries, (2) unseen control flows are observed a long time from execution start.

To resolve this, we provide an optional and low-overhead Indirect Control Flow Target (ICFT) tracer that can be used upfront to augment the statically recovered CFG. Given a set of inputs, it observes concrete executions of the program and records all targets of indirect control tranfers. It then merges information recorded across the different runs, providing the benefits of an entirely dynamic recompiler.

Note that additive lifting complements the ICFT tracer module. Non-deterministic behaviors may lead to certain program paths never being exercised even after extensive tracing, which necessitates sound handling of the unknown control flows in the recompiled binary. As discussed earlier, such behaviors are prevalent in multithreaded machine code because of the various thread interleavings that are possible at runtime.

4.2.3 Callbacks

Correct handling of callback functions is crucial to support multithreaded binaries that use lightweight processes as a threading mechanism (as opposed to user-level threads). This is because the underlying interface of `clone`, which is used to spawn threads on Linux, requires an entry point for the new execution context. These are considered as *external* entry points, since the control flow transfers from an external library to the binary program. Full-program recompilation forces code layout changes, which makes existing code pointers invalid. To ensure correct support for external entry points, recompilers need to identify and rewrite *all* instances of function pointers passed to external procedures precisely. They must also implement the sound handling of the execution context switch from external library code to the recompiled binary and back.

Rev.ng supports external entry points through static linking of libraries and treating them as indirect calls. However, lifting all external library dependencies and linking them into one large program is often undesirable, because it unnecessarily increases binary size and makes it impossible to rely on the system's package manager to update such libraries. McSema and mctoll try to identify function pointer arguments passed to external functions statically to rewrite them. However, tracking pointer-values in machine code, especially if they are passed across function boundaries, can be hard. Also, it may be impossible to solve this problem precisely using static analysis if pointer values are materialized in registers or memory during execution.

BinRec does not rewrite function-pointer arguments and instead inserts trampolines at the original address of function entry points. The trampolines divert control to helpers that marshal native state into emulated state, execute the lifted code and then translate the emulated state back to native before jumping back into external library code. Although this approach is sound, BinRec does not handle the case where the entry point may be executed as part of a different thread. Specifically, it does not correctly initialize the virtual CPU state and the thread-local emulated program stack upon entry, which may cause faults at run time.

External library calls take function pointers as arguments when, (1) performing callbacks, as with `qsort` which requires a user-defined comparator function, (2) spawning new threads of execution, as with `pthread_create` which requires an entry-point in the new execution context. Statically identifying the values of the arguments to such calls is hard, as function pointers could be materialized in registers or loaded from memory at run time. To remain general, recompilers that recover functions must assume that any lifted function could be used as an external entry point.

We insert trampolines at addresses of each of the function starts in the original binary that jump to custom wrappers that enable transition of the execution context from the library code to the lifted code. We implement them to support handling the case when the execution is part of a new thread. This way, irrespective of whether we can statically identify if the original binary spawns new threads, we perform correct recompilation. However, to achieve this, we need to mark all lifted functions as `external` at the IR-level since LLVM could optimize away or inline functions that act as possible external entry points. This increases the overall code size, becazse we need to preserve all function bodies and their callback wrapper implementations during the recompilation process. This approach also prevents inter-procedural compiler optimizations, which affects the performance of the recompiled binary.

To that end, we implement a dynamic analysis based instrumentation pass, on top of the lifted IR, that records the names of functions used as callbacks for a set of inputs. We merge information collected across different runs and subsequently remove wrappers for functions that are not observed as external entry points and mark them as `internal` instead. This makes them available to the compiler for aggressive optimization, which benefits the recompiled output in terms of code size and performance. Note that this is an optional optimization step, and that the recompiled binary provided as input to this stage is a fully functional replacement of the original input.

4.2.4 Per-Thread Stack

Polynima-lifted IR operates on a virtual CPU state that comprises registers, flags and stack memory, that are represented as global variables. For lifted functions, we implement a conservative version of the prototype recovery algorithm as described in Elwazeer et al. in [18]. Functions take as arguments output registers (registers they may read and write to) and input registers (registers they may only read from). All functions only rely on the validity of the stack pointer register passed in as an argument, and do not make any other assumptions about the stack.

To support multithreaded binaries, we mark variables that represent the global state as thread-local to ensure that each thread operates on its own copy of the virtual state. We repurpose the callback wrappers to identify if the binary is in a new thread of execution, and use it to initialize relevant thread-local CPU state such as the segment registers and flags. We allocate memory that acts as the emulated stack for the call-graph starting at the thread-specific entry point, and copy over caller-provided arguments from the native stack into it. The emulated stack pointer then is initialized to point into this allocation.

4.2.5 Handling Atomic Instructions

Support for hardware-provided atomic instructions is necessary to handle multithreaded machine code. A naïve approach to their translation is to decompose them into distinct loads and stores, with all the accesses synchronized using a global lock. This maintains all the guarantees in terms of exclusive access to memory and the ordering of accesses. But, a major drawback is that *all threads* executing an atomic or non-atomic memory accessing instruction have to acquire the same global lock, irrespective of whether the referenced memory locations alias.

To optimize this, we translate atomic instructions to the corresponding compiler built-ins at the LLVM IR-level during lifting. Consider the translation of the `lock cmpxchg dword ptr`

`[rsi], ecx` instruction in Listing 4.1. This instruction compares the value in the `eax` register with the value stored in the destination operand (i.e., memory pointed to by `rsi`). If the two values are equal, the second operand is loaded into the destination. Otherwise, the destination operand is loaded into `eax`. The instruction also updates the zero bit of the `EFLAGS` register depending on the result of the equality. Because the load and store through `rsi` are atomic, we need to hold the lock for the duration of the entire instruction.

We optimize this in Listing 4.2 by using the corresponding `cmpxchg` LLVM IR instruction. Here, we perform the update of the (virtual) `eax` as part of a separate instruction that depends on the result of the `cmpxchg`. By marking the `cmpxchg` as sequentially consistent, we ensure that any synchronization guarantees provided by the ISA are preserved. Since registers are not accessed indirectly, no other thread will race to write to the storage location of the `eax` register.

Listing 4.1: Translation of the `cmpxchg` instruction using non-atomic instructions, protected by a global lock.

```
1  lock(@global_lock)
2  %temp = *%rsi
3  if %eax == %temp:
4    %flags.z = 1
5    *%rsi    = %ecx
6  else:
7    %flags.z = 0
8    %eax     = %temp
9    *%rsi    = %temp
10 unlock(@global_lock)
```

Listing 4.2: Optimization of the translation of `cmpxchg` using the appropriate atomic LLVM instruction.

```
1  %tmp = cmpxchg *%rsi, %eax, %ecx seq_cst
2  %flags.z = %tmp == %eax
3  if !%flags.z:
4    %eax = %tmp
```

4.2.6 Non-Atomic Loads and Stores

Since x86 implements with total store ordering a strong memory model, we need to ensure that the recompiled binary preserves the original program semantics (see also Section 2.2.3). We considered the fence insertion strategy formalized by Lasagne [42]. They formalize a memory model "LIMM" (LLVM IR Concurrency Memory Model) based on fences that are inserted before and after stores and loads, respectively, to prevent memory access reorderings at the IR-level. The fences used in their memory model are designed to replicate the guarantees provided by the ARM ISA in order to be efficient when cross-compiling programs from x86 to ARM. These fences have stronger semantics than acquire/release fences in LLVM IR, but are weaker than sequentially consistent fences. Except for sequentially consistent fences, fences in LLVM IR essentially are required to be paired with another monotonically ordered atomic operation to ensure synchronization. This is unnecessary in the LIMM model, where fences that are not sequentially consistent establish a global happens-before relation between non-atomic memory accesses. They do not integrate their memory-model fully into LLVM, since that would require updating the optimization passes to be aware of the special semantics of these fences. However, their implementation inserts sequentially consistent fence whenever a fence is required. We also noticed they do not mark loads and stores themselves as atomic. This can lead to data-races, since the fences only prevent reordering, but do not provide any guarantees about the atomicity of the memory accesses they surround.

Ignoring the issue of non-atomic memory-accesses, when recompiling x86 programs without the aim cross-compilation, this approach imposes unnecessary overhead. The sequentially consistent fences are lowered to the `mfence` instruction, which is a full memory barrier that prevents all memory accesses from being reordered. The recompiled binary would be correct (assuming all accesses are marked atomic besides inserting the fences), but would impose significantly stronger ordering constraints than the original binary. Instead, we lift every x86 load to a sequentially consistent load instruction, and every x86 store to a store instruction with release semantics. This model ensures that the guarantees of TSO are preserved. The release semantics of stores prevent

loads and stores being reordered past them. The sequentially consistent loads prevent reordering with other loads, and that stores are observed in a consistent order by all threads. For x86, this is a more efficient approach than inserting fences, because sequentially consistent loads and release stores are lowered to the same regular `mov` instructions they were lifted from. It also prevents LLVM from performing incorrect optimizations or removing seemingly unnecessary spin-loops that could change the ordering semantics of the original program.

We note that we have not formally proven the correctness of our translation. However, we have tested our approach on a wide range of multithreaded programs and have observed no issues. Additionally, we have hand-verified our translation using the litmus tests outlined by the x86-TSO memory model [44]. The alternative would be to mark every store as sequentially consistent. This would be the most conservative approach, but would impose unnecessary overhead on the recompiled binary, because sequentially consistent stores are lowered to the `xchg` instruction.

4.3 Evaluation

Our evaluation is guided by the following research questions:

- **RQ1**: Does Polynima make available the transformation infrastructure that is available as part of LLVM to improve and recompile legacy multithreaded binaries?
- **RQ2**: Can we recompile a diverse set of complex real-world multithreaded binaries while maintaining correctness and ensuring a reasonable performance cost?
- **RQ3**: Does our hybrid control flow recovery approach improve state-of-the-art?

4.3.1 Environment and Software

We wrote a wrapper around the radare2 [4] disassembler to output a static (JSON-based) control flow graph representation that includes functions and the basic blocks belonging to them. The ICFT tracer, implemented as a Pin [35] tool, augments this representation with dynamically collected indirect control transfers. We then invoke the translator module, which is built on top of S^{2}E's [16] RevGen [15] utility. This provides us with the infrastructure to translate the individual machine code basic blocks to LLVM IR (LLVM 14). S^{2}E achieves this by first translating machine code to QEMU's TCG intermediate representation and then to LLVM IR. Our translations for atomic instructions are implemented on top of the upstream S^{2}E.

In the lifted IR, we stitch together the lifted basic blocks to create functions based on the recovered control flows. Finally, the rest of our lifting pipeline builds on top of BinRec [3], leveraging passes that enable us to deinstrument the IR emitted by the translator and its infrastructure for lowering the lifted bitcode. Polynima can be accessed through a single command-line utility that provides facilities for project management, disassembly, lifting and (additive) recompilation of binaries. Users need only provide inputs that exercise control flows for the optional dynamic analyses.

We conducted our experiments on a Ubuntu 20.04 LTS system with an Intel i7-8700K CPU running at a base clock of 3.70 GHz, 32 GB RAM, and 6 cores. To ensure stable performance, we disabled frequency scaling, hyper-threading, and frequency boosting. We ran each input five times for performance experiments, summed up their means, and calculated the normalized runtime as a fraction of the baseline. We compiled all binaries with GCC 8, with stack-protector and position-independent execution disabled (`-fno-stack-protector -no-pie`), and optimization level `O3`, except for ConcurrencyKit, which defaults to `O2`.

4.3.2 Comparison with Other Lifters

We tried running other state-of-the-art lifters identified in Liu et al. [32] to lift the binaries that we choose for our evaluation. The authors of RetDec [27] suggest that the tool is designed as a binary lifter, instead of a recompiler, and that the IR is unsuitable for recompilation. Likewise, McSema's [48] authors conveyed that the tool's primary focus is binary lifting and its overall recompilation capabilities are experimental. To evaluate Rev.Ng [17], we used musl-gcc and statically compiled a multithreaded version of the simple "hello world" program. Although we recover a translated binary, we observe faults during execution of the `do_fork` procedure, indicating a lack of support for multithreaded machine code. Lasagne [42], which builds on top of mctoll [53], supports the lifting and recompilation of a subset of multithreaded binaries. However, we could not lift any other binaries apart from those belonging to the Phoenix benchmark suite using their prototype. We also note that the Lasagne was only evaluated on the Phoenix benchmark suite, which does not contain any atomic instructions. This is despite the fact that the key-contribution of Lasagne is a memory model that promises correct lifting of atomic instructions in multi-threaded binaries and cross-compilation to weaker memory models. To our knowledge, Polynima is the **only** binary recompiler that supports real-world multithreaded programs while maintaining original program semantics.

4.3.3 Compatibility and Performance

We test Polynima on a large and functionally diverse set of binaries that comprises real-world utilities and benchmark suites listed in Table 4.1. *We report correct outputs across all the test cases that we run.*

memcached uses pthreads along with compiler built-ins for threading and synchronization. We use the tool memaslap to check the correctness and benchmark the recovered binary performance under load. We run memaslap for 2 minutes with the default configuration of the `get/set` request

Table 4.1: Supported Benchmarks. Lasagne builds on top of mctoll.

Benchmark	LOC	Polynima	Lasagne	McSema	BinRec	Rev.Ng
memcached [50]	24.4k	✓	✗	✗	✗	✗
mongoose [36]	7.4k	✓	✗	✗	✗	✗
pigz [1]	6.4k	✓	✗	✗	✗	✗
LightFTP [24]	2.4k	✓	✗	✗	✗	✗
Phoenix [41]	4.4k	7/7	5/7	0/7	0/7	0/7
gapbs [12]	2.8k	8/8	0/8	0/8	0/8	0/8
CKit (spinloops) [2]	1.3k	11/11	0/11	0/11	0/11	0/11

proportion (0.9/0.1) with 2 and 4 threads in each case. In both cases, the recovered binary reports a less than a 1% difference in the total number of operations performed.

pigz exclusively uses functions provided by pthreads. We benchmark pigz by compressing two files with compression levels fast, default and slow and across the use of 1 / 2 / 4 threads. We observe negligible differences in data processed (in mbs per second) and the total time required for compression in each of the configurations.

We compile **mongoose** the default multi-threaded web-server example to test mongoose which uses pthreads. We configure the siege utility to spawn 25 concurrent threads sending requests to the server for 2 minutes. The average response time for the original server binary is reported to be 2.02s v/s the 2.03s for the recovered one, indicating a minimal performance difference.

For **LightFTP**, which also uses pthreads, we stress test the upload and download speeds for the original and recovered binary. We achieve this by sending concurrent upload and download requests of 1 MB files for ~45 seconds. The average upload times differ by a margin of 2.4% and the download times differ by 9%.

Table 4.2 contains the results for the **Phoenix** benchmark suite, which contains map-reduce style programs that are used to benchmark parallel executions. Phoenix also uses pthreads for synchronization and threading. Since we know the Phoenix benchmarks do not contain any atomic instructions, we can evaluate the impact of missed optimization opportunities when non-atomic

Table 4.2: Performance of Polynima recompiled binaries on the Phoenix benchmark suite. Results in the NA columns report performance if loads and stores are not lifted as atomic instructions.

Benchmark	O0	O0 NA	O3	O3 NA
histogram	0.88	0.81	1.06	1.03
kmeans	0.95	0.54	1.32	1.26
linear_regression	0.90	0.82	1.27	1.20
matrix_multiply	1.01	0.95	1.02	1.02
pca	0.94	0.72	1.13	1.13
string_match	1.27	0.89	1.35	0.94
word_count	1.03	1.03	1.03	1.03
Geomean	0.99	0.81	1.16	1.08

loads and stores are marked as atomic. We use the provided small, medium, and large input datasets to evaluate the performance of the recompiled binaries.

We highlight the performance of Polynima recompiled binaries for the `O0` baseline. For unoptimized binaries, recompiled binaries perform at par or better than the input with an average speedup of $0.99x$, with a maximum speedup of $0.88x$ in the case of *histogram*. However, once we remove the atomicity guarantees for loads and stores, we observe a significant average speedup of $0.81x$. For optimized binaries, we observe a slowdown of $1.16x$, on average, with a maximum slowdown of $1.35x$. When we remove the atomicity guarantees, we observe a smaller slowdown of $1.08x$. In these cases, we observe performance benefits as the compiler, (1) is effective in optimizing the lifted IR, (2) is free to choose SIMD instructions available as part of the underlying hardware for efficient lowering. These results show that Polynima could be useful as a post-release optimizer, for binaries that were originally compiled with little to no optimizations for an older CPU version.

The **gapbs** benchmark suite contains reference implementations of various graph processing algorithms. Programs use OpenMP for parellelization, specifically annotating loop bodies with `#omp parallel` pragmas for concurrent execution. They also use primitives from `std::atomic`, that lower to x86/x64 hardware atomic instructions, for synchronization.

Table 4.3: Performance of Polynima recompiled binaries on the gapbs benchmark suite.

	32-bit		64-bit	
Benchmark	O0	O3	O0	O3
bc	1.17	2.59	1.08	1.40
bfs	0.99	0.97	0.79	0.77
cc	0.74	0.94	0.83	1.32
cc_sv	0.89	1.01	0.82	1.25
pr	1.95	2.90	0.80	1.14
pr_spmv	2.01	2.68	0.72	1.22
sssp	0.72	1.06	0.52	1.08
tc	1.35	1.55	1.35	1.35
Geomean	1.14	1.53	0.83	1.17

We evaluate all gapbs binaries (Table 4.3) on integer inputs, for which we use uniform-random graph inputs of size 2^{20} for each binary. With gapbs, we observe similar trends as with Phoenix. The performances for unoptimized binaries are closer or better than to originals, whereas we observe greater slowdowns for the optimized versions. We observe a disparity in the performance of the recompiled binaries for the 32-bit and 64-bit versions. This is because the 32-bit binaries use the x87 FPU stack for floating-point operations, which are largely emulated in software. The 64-bit binaries, on the other hand, use the SSE registers for floating-point operations, for which we have better support to lower them to the corresponding hardware instructions.

ckit. ConcurrencyKit implements custom concurrency primitives using compiler built-ins (C99) that compile down to use hardware atomic instructions. We first successfully perform correctness checks for all 11 spinlock implementations using the validation test suite. We then use the latency benchmark test as part of the regressions suite to compute the average latency (in terms of the number of clock cycles required) for each spinlock. Each individual test comprises a sequence of lock and unlock operations, executed in a loop. As these involve the lifting and lowering of various hardware atomic instructions, the results help us evaluate our approach to their translation. We

report that the recompiled binary performance is close to the original in almost all cases (Table 4.4), which validates our earlier claims of efficiency and correctness.

Table 4.4: Performance of the original and the recompiled output (in terms of number of clock cycles required) on the latency tests in CKit.

Spinlock	Native	Recovered
ck_anderson	31	25
ck_cas	26	25
ck_clh	26	26
ck_dec	26	24
ck_fas	26	25
ck_hclh	57	57
ck_mcs	56	54
ck_spinlock	26	25
ck_ticket	36	49
ck_ticket_pb	36	35
linux_spinlock	26	23

We use the geometric mean of the results for the unoptimized (`O0`) and optimized (`O3`) Phoenix and gapbs benchmark suites to compute the overall $1.12x$ slowdown. We now discuss the major reasons for degradation in recompiled output performance for optimized binaries (`O3`) in gapbs and Phoenix.

We recompiled the binaries without the stack-symbolization outlined in Chapter 3. Symbolization of multi-threaded programs is not as reliable as for single-threaded programs, and our goal was to generate binaries that are reliable and correct even in the presence of non-deterministic behavior. As seen in the previous chapter, most optimizations in the LLVM ecosystem are designed to work with an IR that distinguishes between program variables. Since we do not recover this for multi-threaded binaries, LLVM has to treat the emulated stack as entirely opaque, which prevents off-the-shelf optimizations from being fully effective.

We also notice the cost introduced by the non-optimal lifting of SIMD instructions and floating point operations. Polynima relies on QEMU [13] helpers to provide translations for such instructions, which are based on emulating them on the virtual CPU state. For certain vector instructions,

LLVM can resynthesize them into intrinsics after lifting, but this translation is not optimal. The performance impact is most visible in the *linear_regression* benchmark, where the core algorithm is implemented as a packed sequence of SIMD instructions in the original binary. Before optimizing vector instructions, this benchmark reported a slowdown of around $10x$ for the `O3` version.

Finally, with OpenMP, each of pragma-annotated loops compile into a distinct function which acts as an entry point into a new thread context. This involves handling a large number of callbacks, which we identify to be another reason for the performance slowdown. Callback-handling includes marshaling of the native registers, copying arguments to the emulated stack, and copying returned registers back to the native state after execution of the lifted function.

We could not reliably recompile most of our benchmark programs with other recompilers. Polynima builds on top of BinRec which outperforms McSema and Rev.Ng recompiled binaries on single-threaded benchmarks [3]. In our result, Polynima performs better on multi-threaded binaries than BinRec on single-threaded ones. Lasagne reports performance results for a subset of binaries from the Phoenix benchmark suite for the downstream task of cross-ISA translation to a different architecture (ARM64), which we do not support yet.

4.3.4 Lifting Time

Overall lift time. We now compare the performance of our control flow recovery approach with that of BinRec and McSema. As neither of the above recompilers supports multithreaded binaries, we apply Polynima to `O3`-compiled binaries from the SPECint 2006 benchmark suite.

For Polynima, we statically collect the CFG and augment it with information from the ICFT Tracer, which is driven with the `ref` inputs for each binary. We ensure the correctness of our control flow recovery process by checking the output of the recompiled binary against the `ref` inputs.

Table 4.5: Lifting Times (in s) the for SPECint 2006 binaries against `ref` inputs and the total number of ICFTs (indirect control flows) recorded in the process

Benchmark	Polynima	BinRec	McSema	ICFTs
401.bzip2	47	69389	3385	21
403.gcc	1380	28468	7378	2350
429.mcf	130	227999	8	0
445.gobmk	634	72307	1063	1241
456.hmmer	427	144529	189	34
458.sjeng	1399	548342	368	69
462.libq.	425	176536	16	0
464.h264ref	1885	65202	586	116
473.astar	265	119436	18	2
483.xalanc.	–	–	17103	–
Geomean	445	137074	238	–

Our prototype could not handle `403.gcc` and `483.xalancbmk` because of failed IR translation for certain superfluous code paths.

We report the total time taken to disassemble, trace, and recompile with Polynima in Table 4.5. We refer to the BinRec paper [3] for relevant numbers for BinRec and McSema. Polynima performs orders of magnitude faster than BinRec while also providing the same precision in terms of the recovered control flow. Also, our performance is comparable to McSema, an entirely static lifter.

To highlight the importance of our hybrid approach, we also report the number of indirect control flows recorded during the tracing process for each program. Consider the case of `429.mcf` and `462.libquantum` that contain no indirect transfers. In such a case, an entirely static approach is efficient and preferable, as the disassembler generated output can be considered precise and complete. However, BinRec performs poorly for both the benchmarks as it needs to trace through the entire program before being able to generate the recompiled output.

For a program such as `445.gobmk`, it is difficult for a static disassmbler to resolve such a large number of indirect control transfers (1241) precisely. Recent work could not functionally verify McSema-recompiled binaries for more than half of the SPEC benchmark suite [32]. Here,

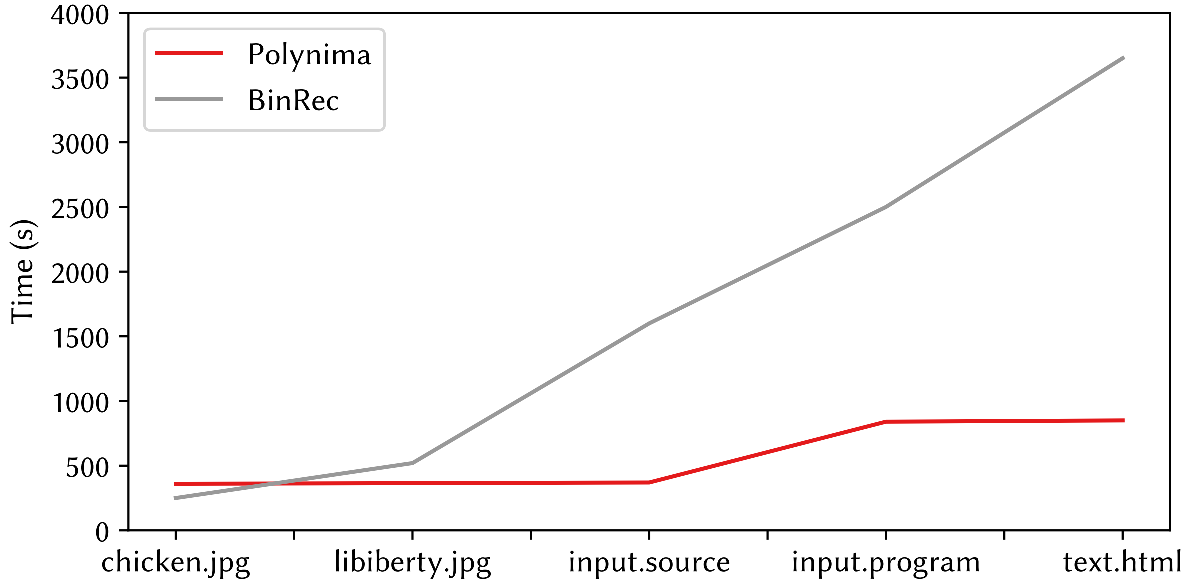

Figure 4.2: Lifting times for BinRec's Incremental lifting v/s Polynima's Additive lifting for `401.bzip2`

Polynima's hybrid approach performs notably better than BinRec, while providing the same precision.

Additive lifting. We lift all of our multithreaded benchmark binaries using additive lifting to test the scalability and robustness of the approach. To evaluate its performance, we compare against BinRec's incremental lifting and report the results in Figure 4.2. We use the `401.bzip2` binary from the SPEC benchmark suite as it was chosen as the demonstrative example in the original paper. We start our measurements by considering a recompiled binary that supports the SPEC *test* inputs. We then measure the time taken (represented by the Y-axis) by both approaches for increasingly complex input files (represented by X-axis).

To summarize, Polynima decouples the process of CFG collection from translating machine code to IR. Performing the IR translation *offline* is key for recompilation to scale to large and non-deterministic binaries. Unlike BinRec, that executes the input program inside a full-scale processor emulator, we run the recompiled output natively. That way, we leverage the relatively low overhead of the recompiled output and do away with the long startup times and emulation cost. Whenever

Polynima discovers a new control transfer, it statically explores the CFG starting at this block and retrofits discovered paths backs into the known CFG. As a result, we see recompilation loops only triggered for *chicken.jpg* and *input.program*, where we explore yet unknown sections of the input CFG.

Chapter 5

Applications

The previous two chapters have developed novel techniques to make lifting binaries easier, reliable, and accurate. This chapter aims to demonstrate the benefits of reconstructing precise program semantics through dynamic analysis by showing how one can perform various analyses and transformations on the lifted IR that are not as easily possible with any other binary recompiler.

5.1 Built-in LLVM Transformations

We demonstrated with Wytiwyg in Section 3.6.2 that lifting and recompilation of legacy binaries can effectively modernize them. Beyond optimization, LLVM provides out-of-the-box a rich set of sanitizers and transformations that can make the binary more secure and resilient to various programming errors and issues introduced by stack symbolization.

One of the most popular sanitizers is AddressSanitizer [43], which detects memory corruption bugs such as buffer overflows and use-after-free errors. The usage of AddressSanitizer is already possible with programs lifted by BinRec without stack symbolization, but the sanitizer is not as effective as it could be with stack symbolization. Since the emulated stack has no structure,

AddressSanitizer does not know the boundaries between distinct stack variables. Without this information, AddressSanitizer cannot detect buffer overflows, use-after-scope and use-after-return errors that relate to incorrect usage of stack variables. However, if Wytiwyg is applied to the binary, AddressSanitizer can be used to its full potential. To our knowledge, there is only one other solution that can detect stack-based memory corruption bugs in binaries effectively, which is MTSan [14]. However, MTSan requires ARM memory tagging support in the hardware, which limits it to analysis of binaries for a single architecture. With Wytiwyg, activating AddressSanitizer is as simple as passing the `-fsanitize=address` flag to the LLVM compiler.

Another popular sanitizer is SafeStack [28], which protects against stack-based memory corruption bugs. SafeStack allocates a separate stack for each thread. Then it determines for each function-local variable whether there can be any accesses to it that could lead to a memory corruption bug. If the object is deemed at risk, it is moved to the separate stack. This way, SafeStack can mitigate against some stack-based memory corruption bugs. Although overflows of stack-allocated arrays are still possible, SafeStack can limit the impact of such bugs. In particular, it can prevent return-oriented programming attacks, because the return address remains on the original stack that holds only variables that are proven to be safe. Again, augmenting a binary lifted by Wytiwyg with is as simple as passing the `-fsanitize=safestack` flag to the LLVM compiler. The authors of BinRec already demonstrated SafeStack in their paper [3]. Since their implementation does not recover any stack variables or functions, the SafeStack instrumentation can only relocate unsafe variables within additional functions linked in during recompilation, but not for any of the lifted code. Nevertheless, this transformation is not required to protect against stack-buffer vulnerabilities overwriting return addresses in lifted code, because return edges in programs lifted with BinRec are already protected by integrity checks.

5.2 Thread Escape Analysis

In Section 4.2.6, we discussed how to preserve the synchronization semantics of regular loads and stores in the lifted IR. Because x86 instructions have TSO semantics, virtually we had to translate loads and stores with sequentially consistent and release semantics, respectively. On x86, the impact of this translation is limited, because the compiler-backend will generate the same code for both types of accesses. However, as we have shown in our evaluation in Section 4.3.3, atomic memory accesses limit possible transformations within the optimization pipeline of the compiler. This issue is even more pronounced when cross-compiling binaries to architectures with weaker memory models, such as ARM. On ARM, a sequentially consistent load is compiled into an `ldar` ("Load-Acquire") instruction, and a release store is compiled to an `stlr` ("Store-Release") instruction. These instructions can be more expensive than regular loads and stores, depending on the micro-architecture.

As mentioned in Section 2.2.3, Lasagne addresses this issue by treating all loads and stores that directly access stack-allocated memory as regular non-atomic loads and stores (in their model, they remove the fences surrounding these instructions). It is easy to construct an example that shows that approach is not sound. Consider the example in Listing 5.1. The program creates a thread that accesses a shared atomic variable `x`. The atomic variable is stack-allocated in the function `main` and passed through a pointer to the thread. At a later point in the program, `main` waits through a spin-loop for the value of `x` to be set to 1. As expected, on x86, this store is compiled to a regular `mov` instruction to the stack slot of `x`. Since the load accesses the stack slot directly, Lasagne would treat this load as a non-atomic load. As with the previous example in Section 2.2.3, this could lead to the spin-loop being optimized away.

A better solution is to perform a thread escape analysis. The goal of this analysis is to determine for each variable whether it can be accessed by a different thread. If it can be proven that the variable is not accessed by a different thread, all accesses to this variable can be converted to

Listing 5.1: Program with stack-allocated atomic.

```
1  #include <atomic>
2  #include <thread>
3  
4  void thread_func(void* arg) {
5      std::atomic_int* x = reinterpret_cast<std::atomic_int*>(arg);
6      // ...
7      x->store(1, std::memory_order_release);
8      // ...
9  }
10 
11 int main() {
12     std::atomic_int x{0};
13     std::thread t{thread_func, &x};
14     // ...
15 
16     // .Lloop:
17     //   mov eax, dword ptr [rsp + 8] // "non-atomic" load from stack
18     //   cmp eax, 1
19     //   jne .Lloop                   // jump back
20     while (x.load(std::memory_order_seq_cst) != 1);
21 
22     // ...
23     t.join();
24 }
```

regular loads and stores. Since this works best when stack variables are symbolized, we modified Wytiwyg to process multi-threaded programs. Since analysis performance of Wytiwyg was not a concern, we protected all accesses to the tracing runtime with a global lock. The other change we had to make was modifying the tracing runtime to be aware of multiple stacks.

To implement the escape-analysis, we mostly relied on facilities already present in LLVM. We inspect all pointers passed to load and store instructions, and check whether they point to a non-escaping stack-allocated object using LLVM's built-in function `isNonEscapingLocalObject`. If the object is not escaping, we can update the instruction to be non-atomic. For example, symbolizing the `kmeans` benchmark reduces the number of number of atomic instructions from 364 to 162. However, this analysis is not perfect. When passing a pointer to a stack-allocated

object to a function, the pointer is converted into an integer using a `ptrtoint` instruction. This means that the escape analysis cannot track the object through function calls, even if the pointer is not escaping within the called function. Generally, a more sophisticated static data-flow analysis would be needed to track the object through function calls and loads and stores to different variables. This analysis also does not need to be limited to stack-allocated objects. Heap-allocated objects allocated through `malloc` and `new` also might not escape a thread and accesses to them could be optimized to be non-atomic.

5.3 Pointer-Identification

The previous section already mentioned that the escape analysis could be more effective if integer could be correctly typed as pointers. This would prevent the escape analysis from losing track of objects passed to functions. However, this is not the only benefit for pointer-identification. Providing correct types for integers that could be pointers is beneficial, because it allows the compiler to perform analyses and optimizations that are targeted at pointers specifically.

Pointer identification also has a significant impact on the recompilation process itself. Recall the example from Listing 2.1 where it was not clear whether the constant operand of a `mov` instruction was a pointer or an integer. If pointers like that cannot be identified and symbolized relative to the base-address of the original binary, the recompiled binary has to map original binary at its original base address in every process. We achieve this with a target-specific linker script that specifies a separate segment for the original binary. Having to map the original binary at its original base address prevents us from compiling binaries as position-independent executables (PIE), which is a security feature that makes it harder for attackers to exploit memory corruption bugs. If all pointers to global variables can be identified, the original binary can be embedded in the recompiled binary's data segment without requiring special handling. Another advantage of pointer-identification would be that if all values are correctly typed if they can hold a pointer, any

kind of pointer-tracking instrumentation, like we did in Wytiwyg, could omit the instrumentation of any non-pointer values. Depending on the program, this could reduce the runtime-overhead of the instrumentation greatly.

As with stack symbolization, pointers need to be identified on a usage-basis. We built a prototype that reuses the infrastructure from Wytiwyg to track values throughout the program. To identify pointers, we first need to identify all base-pointers, i.e., stack-allocated objects, heap-allocated objects, and constants that refer to global variables. The former two are addressed through our stack symbolization and by providing signatures for calls to library functions. To identify constants that refer to global variables, we track all constant integers that lie within the address ranges mapped by the original binary. If a constant, or a value derived from a constant, is observed to be used as the pointer operand to a load or store instruction, we record that. After tracing completes, we replace these constants with pointers relative to the base-address of the original binary.

Identification of global variable pointers is fairly straightforward with dynamic analysis. Now, the second step is to propagate all base pointers through the program. In many cases, this information can be propagated statically in a way that is similar to type inference. This inference can operate in two directions: backwards and forwards. If a value is guaranteed to be a pointer, this information can be forward-propagated to all its uses. Backwards propagation is more complicated. Just because a particular use of a value is a pointer, does not mean that the value itself is a pointer. To not promote false positives, we backwards-propagate the type of a value conservatively. We propagate, if either all uses of a value are pointers, or at least one post-dominating use of that value.

Phi-instructions in loop headers are a special case. A use on the exit path of a loop post-dominates the phi-instruction, but does not post-dominate the uses of the phi-instruction inside the loop. Besides the post-dominating use, we check whether each back-edge of the loop is dominated by a pointer-use. The alternative forward propagation rule is that if all incoming values of the phi-instruction are pointers, we retype the phi itself as a pointer.

After running static propagation, we try to fill in the gaps through dynamic analysis. We instrument the remaining non-pointer instructions in the binary with another simple dynamic analysis that tracks for each value whether it currently holds a pointer. As the program runs, we record for untyped instructions, whether the values they operate on are pointers or integers. After the dynamic analysis completes, we feed the additional information back into our simple inference engine, that propagates and applies types to the remaining values.

5.4 Cross-Recompilation

With the proliferation of new hardware architectures, cross-recompilation can be an important tool to migrate legacy-binaries to these new platforms. Instructing LLVM to generate code for a different target architecture is as simple as setting the lifted module's data-layout to match the one of the target architecture, and passing the name of the target to the compiler when generating an object file. The actual challenge is to ensure compatibility of the lifted IR with the target architecture. For the most part, this means that pointers are required to have the same size within the source and target architectures, and that calls to library functions are converted to match the ABI of the target architecture. Both issues are related to the underlying problem, that performing any changes to the data-layout of data-structures in the lifted program is significantly harder than in the source program. For example, changing the size of a pointer in the lifted program would require selective rewriting all structures that contain pointers, all instructions that compute pointers to fields within these structures, and all constants that encode the size of these structures (e.g., `malloc` or `memcpy`). This limitation not only applies to programs lifted from machine code to IR, but also to programs that were compiled from source code to IR. Nevertheless, binaries that only call library-functions with a consistent ABI across architectures can already be cross-compiled without structural changes to the lifted IR. We have tested this by cross compiling the `mcf` benchmark from the SPECint 2017 benchmark suite from 64-bit x86 to ARMv8. Similarly,

we have cross-compiled the multi-threaded `kmeans` benchmark from the Phoenix benchmark suite from 64-bit x86 to ARMv8.

Chapter 6

Discussion

The previous chapter gave a glimpse into the potential applications of Wytiwyg and Polynima. While they provide a solid foundation for recompilation, there are still limitations and challenges that need to be addressed. This chapter discusses the limitations of our approach and compares it to existing work.

6.1 Binary Compatibility

One of the key concerns for any user of a recompilation tool is whether a binary can be recompiled into a functionally equal binary. The binary emulation approach used to lift binaries to LLVM-IR is highly robust and can be used to recompile a wide range of binaries that are not confined within the same structures as programs compiled from regular C or C++ code. By combining this approach with tracing and online translation of executed instructions to LLVM IR, the original design of BinRec can even recompile some binaries that generate code at runtime. However, recovering higher-level information, such as function boundaries and stack variables as they are represented in LLVM, might simply not be possible if the binary uses obfuscation techniques or

implements different models for structuring control-flow and representing local memory. In such cases, the lifted program can only be analyzed and transformed superficially (at least, without targeted preprocessing). Our goal is to provide a recompilation tool that can be used for a wide range of binaries that fit within the execution models of C or C++, and also conform to some extent to the ABI of the target platform. We aim to support a wide range of legacy software, but not binaries that are specifically designed to be difficult to analyze.

We do not yet support several constructs that can occur in binaries compiled from C or C++ code. We do not handle lifting of the `syscall` instruction. Usage of this instruction in COTS-binaries is uncommon, as software on Linux is usually dynamically linked with the system's standard library (such as `glibc`), which provides higher-level interfaces for system calls. However, some programs might statically link the standard library. For example, programs written in Go rarely use the standard C library, and link the Go runtime statically, which directly invokes system calls. We also do not yet support binaries that use the `longjmp` function from the C standard library, or exceptions in C++.

When lifting x86 programs, we assume that each thread has a single stack that grows downwards and that the stack pointer register of that thread (`esp`/`rsp`) always points to its bottom. Our implementation requires that functions have exactly one entry point, and that control transfers between functions are implemented using `call`- and `ret`-instructions (except for tail calls). Since our approach relies on observing how pointers are used throughout the program, pointer-values need to be "trackable". This means that any operations to derive new pointers from existing ones can be simplified into terms comprising addition and subtraction only. We cannot correctly analyze binaries employing code obfuscation techniques, such as mixed boolean-arithmetic [55], to hide data-flow of pointers. We also do not support binaries with self-modifying code. Additive lifting enables us to recompile binaries with overlapping instructions and obfuscated control flow by design. However, we have not extensively tested our prototype on that capability.

Finally, our implementation currently targets position-absolute ELF binaries without relocation information. Extending our approach to support position-independent binaries is primarily constrained by engineering effort. In fact, binaries linked as position-independent are easier to recompile, because they do not rely on constants to refer to function addresses or global variables. Instead, they use offsets relative to the value of the program counter (e.g., the `rip` register on x86-64), which makes it easy to distinguish between integer constants and pointers.

6.2 Coverage

A primary concern of dynamically driven analyses is attaining comprehensive coverage across the whole program. For Wytiwyg, achieving full coverage encompasses identification of all stack objects and their sizes correctly, and association of all code references with those objects. Albeit our approach yields functional binaries, our evaluation reveals that insufficient coverage results in function layouts that miss some objects, split them, or assign insufficient space to them. At runtime, this can cause out-of-bound accesses with inputs that were not traced. This affects especially variable-sized stack objects (variable-length arrays and C-style `alloca`) as these are converted into allocations of constant size by our implementation. Although this can be partially remedied by augmenting the binaries with `AddressSanitizer` [43], this incurs a significant performance penalty. For practical purposes, such errors are to be treated as incorrect recompilations.

However, previous work suggests that static approaches are plagued by similar problems. As mentioned in Section 3.2, these approaches operate either conservatively (i.e., splitting only if boundaries are provable) or heuristically (i.e., splitting based on assumption made by developers). Particularly complex functions that would benefit the most from local variable symbolization are also the most difficult to process for these tools. Conservative symbolizers are usually incapable of symbolizing such functions, whereas heuristics will fail eventually and lead to a broken binary with no recourse for fixing except manual intervention.

Lifters using static analysis for variable recovery are forced to choose between preserving program functionality and achieving fine-grained variable recovery. The former prevents lifting of the binary to an IR that details precise data-dependencies, inhibiting further processing of the program. The latter is prone to alter program semantics in unpredictable ways, which introduces subtle bugs that are amplified by subsequent optimizations. Wytiwyg provides a path forward in lifting complex programs that exceed the capabilities of static approaches. Using dynamic analysis, complex functions can be symbolized, and we can guarantee that the recompiled programs retain the correct functionality for traced inputs. If new inputs exercise unknown behaviours in the recompiled binaries, the programs can be easily fixed by incrementally reanalyzing it. Further, in the scope of this work, we consider Wytiwyg purely in a vacuum. In practice, our approach could be combined with a robust, heuristics-based static analysis. Such an integration would not only provide the same functional guarantees, but would also minimize issues caused by insufficient coverage.

We already demonstrate that approach in Polynima, where we use a combination of static and dynamic analysis to lift multi-threaded binaries. Ideally, Polynima would lift binaries primarily using dynamic analysis, and use static analysis to fill in the gaps. This way, Polynima benefits from the availability of extensive test-suites for known programs and advancements in techniques such as fuzzing and symbolic execution that recover coverage-inducing inputs for unknown binaries.

6.3 Type-Level Rewriting

We have demonstrated how to recover variables and objects from the stack of a binary using dynamic analysis. These objects are devoid of any type information. As mentioned in Section 5.4, this has only limited impact for reoptimization and recompilation when targeting the same platform as the original binary. However, any modification to the program that changes the ABI

or the data layout of objects (e.g., adding or removing fields from a struct) is not possible with our current solution.

There is a significant body of work on recovering type information from binaries [30], [31], [46], [47], [54]. These approaches aim to aid binary analysis and reverse engineering efforts, which limits their applicability to recompilation. Even when type information for variables is fully available and precise, symbolizing accesses through computed pointers in terms of these types is a non-trivial problem. We have not yet explored how to recover type information for objects in binaries in a way that is suitable for recompilation and leave this for future work.

Chapter 7

Conclusion

Binary recompilation is a powerful technique for transforming legacy software into a form that is usable with modern tools and techniques. Existing state-of-the-art recompilers relying purely on static analysis, heuristics, and manual intervention have been known to produce binaries that are prone to correctness issues and cannot be effectively reoptimized. Fully dynamic recompilers can produce correct binaries, but they are often slow, provide insufficient coverage, and cannot effectively reoptimize the lifted binaries as well. To address these limitations, this thesis has proposed two novel binary recompilation techniques that leverage existing static and dynamic recompilation capabilities to improve lifted binaries through advanced dynamic analyses.

We have presented Wytiwyg, the first binary recompiler capable of transforming COTS-binaries to a compiler-level IR while recovering stack symbols accurately. By leveraging dynamic analyses to recover source-level structures and iteratively refine lifted binaries, we surpass the limitations of existing recompilers that depend on manually tuned heuristics. Our evaluation demonstrates that the fine-grained partitioning of stack variables allows compilers to reoptimize binaries effectively and achieve considerable speedups compared to previous state-of-the-art solutions. These results

highlight the importance of precise stack variable recovery to lifting binaries to an IR that is useful for downstream applications, and that Wytiwyg is highly effective.

Since non-determinism is a key limitation of dynamic approaches, we have also presented Polynima, the first practical binary recompiler for multithreaded and non-deterministic x86/x64 binaries. We designed a hybrid control flow recovery approach that combines the benefits of static and dynamic techniques while also providing an efficient strategy for handling control-flow misses. Polynima was evaluated on a wide range of real-world utilities and benchmark suites, demonstrating its ability to lift complex multithreaded machine code correctly. The combined results of these contributions provide a pathway for future research in combining static and dynamic analyses to lift complex real-world binaries.

Bibliography

[1] Mark Adler, *pigz*, Software, Accessed: 2023-10-02. [Online]. Available: http://zlib.net/pigz.

[2] Samy Al Bahra, *ConcurrencyKit*, Software, 2011. [Online]. Available: https://github.com/concurrencykit/ck (visited on 10/02/2023).

[3] Anil Altinay *et al.*, "BinRec: Dynamic binary lifting and recompilation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827. DOI: 10.1145/3342195.3387550.

[4] Sergi Alvarez, *Radare2: Libre reversing framework for Unix geeks*, Accessed: 2023-10-02, 2006. [Online]. Available: https://github.com/radareorg.

[5] Jim Alves-Foss and Jia Song, "Function boundary detection in stripped binaries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, pp. 84–96, ISBN: 9781450376280. DOI: 10.1145/3359789.3359825.

[6] Kapil Anand *et al.*, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 295–308, ISBN: 9781450319942. DOI: 10.1145/2465351.2465380.

[7] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos, "An In-Depth analysis of disassembly on Full-Scale x86/x64 binaries," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, pp. 583–600, ISBN: 978-1-931971-32-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse.

[8] Dennis Andriesse, Asia Slowinska, and Herbert Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, New York, NY, USA: IEEE, 2017, pp. 177–189. DOI: 10.1109/EuroSP.2017.11.

[9] Apple Inc., *OS X ABI Mach-O file format reference*, Feb. 4, 2009. [Online]. Available: https://developer.apple.com/library/mac/documentation/developertools/conceptual/MachORuntime/Reference/reference.html (visited on 09/04/2014).

[10] Arm Limited, *Arm® compiler for embedded, User guide*, version 6.20, 2023.

[11] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics.," in *Network and Distributed Systems Security (NDSS) Symposium 2018*, (Feb. 18–21, 2018), San Diego, CA, USA: The Internet Society, Feb. 2018, ISBN: 1-1891562-49-5. DOI: 10.14722/ndss.2018.23300.

[12] Scott Beamer, Krste Asanović, and David Patterson, *The GAP benchmark suite*, 2017. arXiv: 1508.03619 [cs.DC].

[13] Fabrice Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[14] Xingman Chen *et al.*, "MTSan: A feasible and practical memory sanitizer for fuzzing COTS binaries," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 841–858, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/chen-xingman.

[15] Vitaly Chipounov and George Candea, "Enabling sophisticated analyses of ×86 binaries with RevGen," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, New York, NY, USA: IEEE, 2011, pp. 211–216. DOI: 10.1109/DSNW.2011.5958815.

[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, CA, USA: Association for Computing Machinery, 2011, pp. 265–278, ISBN: 9781450302661. DOI: 10.1145/1950365.1950396.

[17] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta, "Rev.Ng: A unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017, Austin, TX, USA: Association for Computing Machinery, 2017, pp. 131–141, ISBN: 9781450352338. DOI: 10.1145/3033019.3033028.

[18] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua, "Scalable variable and data type detection in a binary rewriter," 6, vol. 48, New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 51–60. DOI: 10.1145/2499370.2462165.

[19] *Executable and Linking Format (ELF) specification*, Tool Interface Standard (TIS), 1995.

[20] Free Software Foundation, Inc., *GNU Compiler Collection (GCC) internals*, version 13.2, Jul. 27, 2023. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/.

[21] Peter Goodman and Akshay Kumar, *Lifting program binaries with McSema*, Presented at *9th International Summer School on Information Security and Protection* (Canberra, AU) *(ISSIP '18)*, 2018.

[22] Andrea Gussoni, Alessandro Federico, Pietro Fezzardi, and Giovanni Agosta, "Performance, correctness, exceptions: Pick three," in *Binary Analysis Research Workshop 2019*, ser. BAR 2019, San Diego, CA, USA: The Internet Society, 2019, ISBN: 1891562584. DOI: 10.14722/bar.2019.23093.

[23] HexRays, *IDA Pro*, 1991. [Online]. Available: https://www.hex-rays.com/ida-pro (visited on 2023).

[24] hfiref0x, *LightFTP*. [Online]. Available: https://github.com/hfiref0x/LightFTP (visited on 10/02/2023).

[25] R. Nigel Horspool and Nenad Marovac, "An approach to the problem of detranslation of computer programs," *The Computer Journal*, vol. 23, no. 3, pp. 223–229, Aug. 1980. DOI: 10.1093/comjnl/23.3.223.

[26] ISO/IEC, *ISO/IEC 14882:2020, Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2020.

[27] Jakub Křoustek, Peter Matula, and Petr Zemek, "Retdec: An open-source machine-code decompiler," presented at the Botconf 2017, 2017.

[28] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song, "Code-Pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 147–163, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

[29] Chris Lattner and Vikram Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, New York, NY, USA: IEEE, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[30] JonHyup Lee, Thanassis Avgerinos, and David Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Network and Distributed Systems Security (NDSS) Symposium 2011*, San Diego, CA, USA: The Internet Society, 2011.

[31] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu, "Automatic reverse engineering of data structures from binary execution," in *Network and Distributed Systems Security (NDSS) Symposium 2010*, San Diego, CA, USA: The Internet Society, 2010.

[32] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao, "Sok: Demystifying binary lifters through the lens of downstream applications," in *2022 IEEE Symposium on Security and Privacy (SP)*, New York, NY, USA: IEEE, 2022, pp. 1100–1119. DOI: 10.1109/SP46214.2022.9833799.

[33] LLVM, *LLVM language reference manual*, version 17.0, 2024. [Online]. Available: https://llvm.org/docs/LangRef.html.

[34] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper, *The GNU C library reference manual*, version 2.39, 2024. [Online]. Available: https://sourceware.org/glibc/manual/2.39/pdf/libc.pdf.

[35] Chi-Keung Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2005. DOI: `10.1145/1065010.1065034`.

[36] Sergey Lyubka and Cesanta Software Limited, *Mongoose — embedded web server*, Accessed: 2023-10-02, 2004. [Online]. Available: `https://mongoose.ws`.

[37] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell, *System V application binary interface — AMD64 architecture processor supplement (draft)*, version 0.99.6, 2013.

[38] Microsoft Corporation, *PE format*, Aug. 18, 2023. [Online]. Available: `https://learn.microsoft.com/en-us/windows/win32/debug/pe-format` (visited on 10/14/2023).

[39] National Security Agency, *Ghidra*, Mar. 5, 2019. [Online]. Available: `https://ghidra-sre.org`.

[40] Chengbin Pang *et al.*, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 833–851. DOI: `10.1109/SP40001.2021.00012`.

[41] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, IEEE, 2007, pp. 13–24. DOI: `10.1109/HPCA.2007.346181`.

[42] Rodrigo C. O. Rocha *et al.*, "Lasagne: A static binary translator for weak memory model architectures," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 888–902, ISBN: 9781450392655. DOI: `10.1145/3519939.3523719`.

[43] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, Boston, MA, USA: USENIX Association, 2012.

[44] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010, ISSN: 0001-0782. DOI: `10.1145/1785414.1785443`.

[45] Hovav Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561, ISBN: 9781595937032. DOI: `10.1145/1315245.1315313`.

[46] Yan Shoshitaishvili *et al.*, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, New York, NY, USA: IEEE, 2016, pp. 138–157. DOI: `10.1109/SP.2016.17`.

[47] Asia Slowinska, Traian Stancescu, and Herbert Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Network and Distributed Systems Security (NDSS) Symposium 2011*, San Diego, CA, USA: The Internet Society, 2011.

[48] Trail of Bits, *McSema*, Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode, version 0.2.0, Jan. 30, 2015.

[49] Trail of Bits, *Remill*, Library for lifting machine code to LLVM bitcode, version 2.0.0, Dec. 4, 2017.

[50] Anatoly Vorobey, Brad Fitzpatrick, and Danga Interactive, *Memcached*, Accessed: 2023-10-02, 2003. [Online]. Available: `https://memcached.org`.

[51] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl, "From hack to elaborate technique–a survey on binary rewriting," *ACM Comput. Surv.*, vol. 52, no. 3, Jun. 2019. DOI: `10.1145/3316415`.

[52] David Williams-King *et al.*, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 133–147, ISBN: 9781450371025. DOI: `10.1145/3373376.3378470`.

[53] S. Bharadwaj Yadavalli and Aaron Smith, "Raising binaries to llvm ir with mctoll (wip paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 213–218, ISBN: 9781450367240. DOI: `10.1145/3316482.3326354`.

[54] Zhuo Zhang *et al.*, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy (SP)*, New York, NY, USA: IEEE, 2021, pp. 813–832. DOI: `10.1109/SP40001.2021.00051`.

[55] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *Information Security Applications*, Sehun Kim, Moti Yung, and Hyung-Woo Lee, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 61–75, ISBN: 978-3-540-77535-5. DOI: `10.1007/978-3-540-77535-5_5`.

Appendix A

Example Programs

A.1 Overlapping Instruction Sequence

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

int add5(int val, uintptr_t target);
__asm__(
    ".p2align 4\n\t"
    ".globl  add5\n\t"
    ".type   add5, @function\n"
"add5:\n\t"
    "movl  $0xbb0eeb00, %eax\n\t"
    "movl  $0x5000000, %ecx\n\t"
    "addl  %ecx, %eax\n\t"
    "jmpq  *%rsi\n\t"
    "leal  5(%rdi), %eax\n\t"
    "retq\n\t"
    "movq  secret@GOTPCREL(%rip), %rsi\n\t"
    "jmpq  *%rsi");

int secret()
{
```

```
    puts("Secret message!");
    return 42;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <value>\n", argv[0]);
        return 1;
    }
    uintptr_t target = (uintptr_t)add5 + 0xe;
    if (strcmp(argv[1], "password") == 0)
        target -= 12;
    int result = add5(atoi(argv[1]), target);
    printf("Result: %d\n", result);
    return 0;
}
```