UNIVERSITY OF CALIFORNIA,
IRVINE

Practical Recompilation of Multithreaded Binaries: Choreographing Static and Dynamic Techniques

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Chinmay Diwakar Deshpande

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ardalan Amiri Sani
Professor Qi Alfred Chen
Dr. David Gens

2024

DEDICATION

This work is dedicated to आम्मू आणि बाबा. And to all those who stood by my side in this journey.

"gg ez"

*Often said at the end of a long and arduous, but victorious, endeavor; with the **ez** emphasizing that the author did not really need to try hard to achieve the win (when in fact, they were close to losing)*

Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

This section serves to convey my gratitude to all those who have been there for me over the years. I have tried my best, but I am sure to have inadequately represented the impact and contributions of so many. For that, I apologize.

First and foremost, I would like to thank my advisor, Professor Dr. Michael Franz. Thank you, Michael, for allowing me to be a part of and conduct research at the Secure Systems and Software Lab (SSLLab). I am forever grateful for the academic guidance and mentorship that I have received over the past 5 years. More so, thank you for teaching me to do the right thing. Michael's unwavering trust in me was a beacon of hope when the going got tough.

I was extremely fortunate to have worked with some fantastic PostDocs at SSLLab. I picked up so much on how to tackle hard problems and go about conducting meaningful research through them. Thank you to Dr. Yeoul Na, Dr. Adrian Dabrowski, and Dr. Felicitas Hetzelt. Special thanks go out to Dr. David Gens, for answering my silly questions, for helping me get my first publication and now, for serving on my committee. Thank you, David, I have learned so much from you.

I would also like to thank our lab alumni – Dr. Dokyung Song, Dr. Anil Altinay, Dr. Joseph Nash, Dr. Taemin Park, Dr. Alexios Voulimeneas, and Dr. Paul Kirth – who have been a pleasure to interact and work with. I am particularly indebted to Dr. Prabhu Rajasekaran, who has been a mentor to me. Prabhu convinced me to join UCI and was always there to help me make difficult career choices.

There is no doubt that SSLLab nurtures some of the best research talent – in terms of creativity and hard work. But our lab feels and operates like a family, which I am more thankful for. Thank you to Dr. Min-Yih, Mitchel, Matthew, Dr. Fabian, André, Hongyu, Tianjiao, Billy (Po-An), Nicholas, Mahbub, Weitao, James, Nathaniel and Allan. Mitchel Dickerson is one of the kindest and warmest human beings I've ever met. Thanks, Mitch, for never letting me feel like I'm away from home, for hosting board game nights, and for the companionship. I've come across a lot of smart and talented people, but few as sharp as Dr. Fabian Parzefall. Thank you Fabian for inspiring me to strive hard towards delivering the highest quality of work, for the free-flowing discussions, and for being such a good friend. I'm so proud of what we've achieved together. Hongyu, thank you for being such a good friend. You're the sweetest and most hardworking person, and I wish only the best for you. Thanks Erika for the ever-so-thoughtful questions and for being so welcoming.

My time in Irvine was immensely enjoyable due to friends I made here. Thank you Sakshi for the fun dinner nights at 555. Shoutout to the S&T (full name redacted) group – Romit, Ritwick, Prasad and Siddhant – for the unadulterated nonsense that helped me destress during the PhD. Nimish, I am grateful for your invaluable friendship and for instilling confidence in me when I felt low. I dont know why, but I always felt that you believed in me more than anyone else. Thanks for that, sir. Neil, thank you for being such an awesome roommate and friend. I truly cherish the long walks and deep discussions we had during your stay at 555, and beyond. Aman, thanks for teaching me how to deal with stress and smile through the ups and downs of life. I'm going to

miss our deep chai-pe-charcha sessions. Shruti and Priya thank you for making my weekends all the more cheerful.

The 6A gang from NITK is near and dear to my heart. Thank you – Dr. Amey, Giri, Sutej, Kishor, Palash, Pratheek, Rajat, Srinidhi, and Sebin – for the laughs, the memes, but crucially, the friendship. Abraham and Harish, thanks for making the trip to attend my defense, it was heartwarming to see your faces in the crowd. Siddharth and Shiva(prakash), thank you to the both of you for being so incredibly supportive and motivating me along the journey. Sagar, thank you for the times we spent in 13A (especially the nights when the moon was shining) and for giving the right advice when I most needed it. It is said that one should have a "3 AM friend", someone you can rely on to always be there for you. That, for me, was Kunal Jha. Kunal, there's so much to thank you for, but I do not have enough words at my disposal. I owe you everything buddy.

Thank you to the No Internet Access (NIA) team – Kishor, Karthik, Ajith and Sushant (our El Capitan) – for introducing me to the intriguing world of computer security. I will forever treasure the weekends spent in Sushant's room poring over obscure CTF problems and consuming absurd amounts of (bad) coffee. Thank you, Anton Kochkov (XVilka), for your mentorship during my days with radare2.

Thank you to my summer internship mentors – Peter LaFosse, Ryan Snyder, Jordan Wiens (Vector35), Dr. Nathan Chong, Dr. Daniel Schwartz-Narbonne (AWS), Dr. Nilo Redini, and Murali Somanchy (Qualcomm). I have become a better engineer and researcher due to your insights and guidance. Special thanks to Sumit Lohani (McAfee), for teaching me how to tackle life and work with a positive attitude.

Thank you Keval for the decades of friendship. Sailee, thank you for being you. You inspire me to think hard, challenge myself, and become a better individual.

Kauts, Vishal, Prithiv, Sagar and Keith, thanks for all the fun times in Acme Complex. My trips to India are always a joy because of your lot.

I also want to thank the developers of the amazing video game Dota2, which was available to me as an escape mechanism from the stresses of the PhD.

Thank you to my committee members Dr. Ardalan Amiri Sani and Dr. Qi Alfred Chen, for their valuable inputs and assessment of my work.

Finally, but most importantly, I would like to thank my family for their unconditional love and support. My Father, Diwakar Deshpande, for being the stable rock in my life and for his unfaltering belief in me to achieve success. My Mother, Geeta Deshpande, for always having my back and for raising me to be the person that I am today. I would not be here without them. I cannot convey how grateful I am to be a part of their life. Here's wishing them the best of health and happiness.

This material is based upon work partially sponspored by the Defense Advanced Research Projects Agency (DARPA) under contract N66001-20-C-4027, W31P4Q20-C-0052, 140D04-23-C-0063 and

140D04-23-C-0070 and by the United States Office of Naval Research (ONR) under contracts N00014-17-1-2782, N0001422-1-2232 and N00014-21-1-2409. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ONR or DARPA.

Portions of Chapter 3 are a reprint of the material as it appears in "Polynima - Practical Hybrid Recompilation for Multithreaded Binaries", used with permission from ACM (CC BY). The co-authors listed in this publication are Fabian Parzefall, Felicitas Hetzelt and Michael Franz.

Portions of Chapter 4 are a reprint of the material as it appears in "StackBERT: Machine Learning Assisted Static Stack Frame Size Recovery on Stripped and Optimized Binaries", used with permission from ACM (CC BY). The co-authors listed in this publication are David Gens and Michael Franz.

Portions of Chapter 5 is a reprint of the material as it appears in "What You Trace is What You Get: Dynamic Stack-Layout Recovery for Binary Recompilation", used with permission from ACM (CC BY). The co-authors listed in this publication are Fabian Parzefall, Felicitas Hetzelt and Michael Franz.

Michael Franz directed and supervised research which forms the basis for the dissertation.

VITA

Chinmay Diwakar Deshpande

EDUCATION

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

Bachelor of Technology in Information Technology — **2017**
National Institute of Technology Karnataka — *Surathkal, India*

EXPERIENCE

Graduate Research Assistant — **2019–2024**
University of California, Irvine — *Irvine, California*

Interm Engineering Intern — **Summer 2023**
Qualcomm Inc. — *San Diego, California*

Applied Scientist Intern — **Summer 2021**
Amazon Web Services Inc. — *Minneapolis, Minnesota*

Summer Intern — **Summer 2020**
Vector35 Inc. — *Melbourne, Florida*

Software Development Engineer — **2017–2019**
McAfee Software Ltd. — *Bangalore, India*

TEACHING EXPERIENCE

Teaching Assistant — **2019, 2020, 2023**
University of California, Irvine — *Irvine, California*

Student Mentor — **2014–2017**
Avanti Fellows — *Mudipu, India*

REFEREED CONFERENCE PUBLICATIONS

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).

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).

REFEREED WORKSHOP PUBLICATIONS

Chinmay Deshpande, David Gens, and Michael Franz, "StackBERT: Machine Learning Assisted Static Stack Frame Size Recovery on Stripped and Optimized Binaries." *In 14th ACM Workshop on Artificial Intelligence and Security* (AISec '21).

ABSTRACT OF THE DISSERTATION

Practical Recompilation of Multithreaded Binaries: Choreographing Static and Dynamic Techniques

By

Chinmay Diwakar Deshpande

Doctor of Philosophy in Computer Science

University of California, Irvine, 2024

Professor Michael Franz, Chair

Recompiling legacy programs from source can be challenging due to outdated build environments, the unavailability of original source code, or lacking vendor support. Without access to the compiler ecosystem, it becomes impossible to reoptimize legacy software for modern hardware, apply program modifications, or utilize compiler-level hardening features. This limitation is particularly costly when addressing critical security vulnerabilities that necessitate immediate patching.

Binary recompilation aims to bridge this divide by "lifting" binary executables to a compiler-level intermediate representation (IR) and "lowering" them back again. Recompilers hold the promise of making available the rich analysis and transformation ecosystem of the compiler, that is typically used with source-level programs, to modify and reoptimize binary programs. But, state-of-the-art tools in this space have not seen much adoption due to practical concerns.

Current recompilers are entirely static or dynamic in their approach to recompilation. Static tools are fast, but rely on heuristics for reasoning about possible control flow paths in the program and recovering stack-local variable information. Transformations that rely on unsound analyses may lead to divergences from the original program behavior or even faults during recompiled binary execution. Moreover, static recompilers impose unnecessary size and performance overheads on

the recompiled output due to their conservative approach to lifting. Dynamic techniques, on the other hand, offer precision but are inefficient due to their high tracing overheads. Crucially, none of these tools generally handle multithreaded binaries, that are ubiquitous in the modern software space.

In this dissertation, we improve the state-of-the-art in binary recompilation along two complementary directions by composing static and dynamic techniques. First, we present Polynima, the first practical recompiler that is able to reliably lift and recompile complex real-world multithreaded binaries and benchmark suites. We demonstrate the usefulness of our tool by fixing a critical synchronization issue in an off-the-shelf FTP server binary by leveraging the compiler-level pass infrastructure. Polynima implements a novel, hybrid control flow recovery strategy that combines the benefits of static and dynamic lifters, along with soundly handling the inherent non-determinism in multithreaded programs. Finally, we design a dynamic procedure to detect implicit synchronization primitives in binaries and use that to improve overheads induced due to the conservative handling of shared memory accesses in the lifted IR.

Since maintaining an implicit call stack is central to most modern programming languages and their compilers, soundly reasoning about stack memory layouts is central to correct and efficient binary recompilation. We tackle this problem through two novel approaches: StackBERT and WYTIWYG. StackBERT is a learning-based technique to recover function frame sizes in binary programs. We show the inadequacy of current static approaches to reason about the program stack in binaries and provide evidence that our trained models are valuable for target architectures that are largely unsupported by existing analysis tools. WYTIWYG significantly improves upon this by designing instrumentation-based dynamic analyses that recover program variables. Our fine-grained approach tracks the dataflow across stack memory operations at runtime and infers bounds that helps delineate distinct objects. By using this information to refine the lifted IR, our findings show that WYTIWYG makes it possible to leverage the full potential of the compiler ecosystem to recompile and reoptimize legacy binaries.

Chapter 1

Introduction

1.1 Overview

The maintenance of software distributed as binary executables becomes challenging over time. Due to obsolete build environments, unavailability of the original source code or the lack of vendor support, recompiling legacy programs from source may not be possible. This denies the users of such programs the substantial advancements in modern compiler technologies of the past few decades. The lack of access to the compiler ecosystem prevents reoptimizing legacy software for modern hardware, apply program modifications and leverage compiler-level hardening measures. This limitation can be particularly costly when dealing with criticial security vulnerabilities that require immediate patching. As replacing such legacy software can be prohibitively expensive or even infeasible, users may be required to explore ways to work directly with binary programs.

Binary Rewriting techniques aim to analyze and modify a binary program without requiring access to its source code. They enable the application of various program transformations – such as those relating to instrumentation, fuzzing, and security – to commercial off-the-shelf binaries. ***Binary Recompilation*** is a specific rewriting technique that "lifts" binaries to a compiler-level

intermediate representation (IR) with the goal of "lowering" it back down again using the existing compiler infrastructure. This differs from other rewriting approaches as they use IRs that are often limited in scope and focused on niche transformations, such as control flow and instruction-level modifications. Fundamentally, recompilation aims to make available the rich analysis and transformation ecosystem of the compiler, that is typically used with source-level programs, to modify and reoptimize binary programs. State-of-the-art recompilers [12, 14, 40, 41, 115] target LLVM IR [69] due to its active community, modular design and the rich tooling support. However, most of these tools have not seen widespread adoption due to two major practical concerns, (1) the reliance on exclusively static or dynamic approaches limits recompilation to be either sound or efficient, and (2) the failure to handle complex multithreaded binary programs limits the scope of their practical application.

Precise control flow recovery is an operational necessity for the application of program-wide transformations on binaries. Failure to do so may lead to imprecise code pointer relocations and, therefore, run time crashes in the recompiled binary while realizing such paths. In the general case, this is an undecidable problem [56] as compilers often do not emit any labels that help differentiate code and data bytes. Existing recompilers are are either *entirely* static [14, 40, 41, 115] or dynamic [12] in their approach toward control flow recovery. Static instruction recovery is fast but employs imprecise heuristics to predict targets of indirect control transfers [89]. Dynamic recompilers [12], on the other hand, analyze concrete executions of the target program, enabling them to handle precise instruction recovery and indirect control flows by design. The downside is that their approach is inefficient due to the high tracing overheads imposed to support a sufficient subset of the original binary's functionality.

With the ubiquity of multicore processors, programs are designed to fully leverage the benefits of the underlying hardware which is frequently achieved through multithreading. A central concern for modern recompilers is the support for multithreaded binaries, as they introduce an entirely new class of issues. First, such programs may exhibit behaviors that depend on the spe-

cific sequence of thread interleavings encountered at runtime. Therefore, approaches that claim to support multithreaded binaries must implement a sound strategy to handle unknown control flows seen during recompiled binary execution. After lifting to IR, the compiler is free to reorder shared memory accesses during optimizing transformations, as the binary does not include any synchronization information. This may lead to erroneous and divergent program outcomes that break original program semantics [98]. As a result, existing approaches that insert memory fences to prevent such reorderings may be overly conservative and therefore impede IR-level optimizations [22]. Besides, to fully support multithreaded binaries, recompilers must also correctly handle constructs such as hardware-supported atomic instructions, callback functions and most importantly, the per-thread program stack. Unfortunately, *none* of the current recompilers address the above challenges.

Compiler IRs such as LLVM IR [69] are designed to work with abstractions that accurately represent source-level constructs. One such key construct is the program stack that hosts local variables, function arguments, saved registers and spills. For the lifted IR to be amenable to off-the-shelf transformations and optimizations, it is necessary to correctly recover and model the stack. While binary format standardization and debugging information can help solve this problem to some extent, they may not always be available, sufficiently precise, or even correct [19, 106] – particularly in the context of heavy compiler optimizations. This is why most state-of-the-art approaches currently resort to emulated stack environments for lifted binaries [12, 40, 41]. However, this leads to the recompiled programs being slower than the original, even after the application of aggressive optimizations [12].

Recompilers may choose to "refine" the stack memory representation in the IR at two broad levels of granularity to solve this problem. A coarse-grained approach involves lifting stack frames for individual functions, where the fundamental challenge is to recover bounds on the maximum frame size. However, this can be particularly hard due to non-uniform function inlining or if the frame size is dependent on runtime behaviors. In fact, a function's stack frame may not be

bounded at all and programs that call `alloca` (or one of its many variants) with dynamically determined size argument or make use of Variable Length Arrays (VLAs) can exhibit differently sized stack frames for different inputs. Even if frames are conservatively recovered in the IR, the efficacy of many program analysis techniques is severely diminished without the lack of variable information. A more fine-grained approach further splits the per-function frame into distinct objects that loosely represent source-level program variables. This is also difficult, as it requires precisely identifying and translating (as part of the IR), *all* references to stack memory that denote distinct objects. Some recompilers implement a procedure to recover such a mapping, but they either rely on unsound heuristics [41] or are heavily conservative in their approach [14, 44]. Importantly, all of these tools are fully static, which also introduces an inherent imprecision [56].

1.2 Contributions and Outline

This dissertation details novel contributions that improve the state-of-the-art in binary recompilation.

Chapter 2 develops the necessary background on binary recompilation. We describe the various subproblems involved in the process of lifting machine code to IR, refining and lowering the IR to generate a recompiled binary.

Chapter 3 describes Polynima, the first practical binary recompiler that is able to reliably lift and recompile complex multithreaded binaries. Polynima implements a novel, hybrid control flow recovery strategy, *additive lifting*, that combines the benefits of static and dynamic lifters. Then, we design novel instrumentation-based dynamic analyses that refine the lifted IR. Specifically, we construct a way to detect implicit synchronization primitives in binaries and use that to improve the performance of the recompiled output.

Chapter 4 describes StackBERT, a learning based architecture-agnostic approach to recover in-

dividual function frame sizes in binary programs. We show that the problem of inferring the maximum bound on frame sizes in binaries is hard and then detail how current static approaches are lacking. Then, we demonstrate the efficacy of our trained models with respect to this problem from the binary function body alone - reporting a 93.44% validation accuracy on standard benchmarks that were not seen during training.

Chapter 5 details WYTIWYG, a unique instrumentation-based dynamic analysis that precisely recovers stack objects in the lifted IR. First, we track the dataflow of stack memory operations at runtime. Based on the collected information, we implement an analysis that infers bounds to delineate distinct stack objects and use that to refine the IR. WYTIWYG makes it possible to leverage the full potential of the existing compiler ecosystem to analyze, transform and reoptimize legacy binaries.

Chapter 6 provides a discussion of the inherent security - correctness gap in binary recompilation and planned future work. Here, we outline how known problems in compiler correctness research manifest in the context of recompilation.

Chapter 7 concludes this dissertation by highlighting our key contributions.

Chapter 2

Background

2.1 Binary Rewriting

Binary rewriting describes the general process of making modifications to the binary and producing a new rewritten binary. The eventual goal of the transformations could be instrumentation, hardening, optimizations, or deobfuscation [111]. In the recent years, industry and academia have contributed to the growth of interest in rewriting for a wide range of downstream applications that include security [64], optimization [88], cross-ISA translation [16], and debloating [9, 96].

At a high level, rewriters can be classified as static or dynamic.

Static rewriters operate on the binary while it is stored in persistent memory. Static techniques range from being *direct* to *minimal-invasive* to *full-translation* [111]. Minimal-invasive (and direct) schemes target specific tasks such diverting the control flow, inserting trampolines, or performing instruction-level modifications. The idea is to modify as few bytes of the binary program as possible, while achieving the desired outcome. Although this results in an efficient process and a performant rewritten binary, the overall transformation capabilities of such tools are typically

limited as they do not recover source-level abstractions.

Full-translation techniques, on the other hand, usually translate programs to specialized IRs and *reassemble* a new binary. IRs used by such rewriters aim to faithfully represent the original program semantics in an architecture agnostic manner. Examples include VEX IR [108], Binary Analysis Platforms's (BAP) [29] IL, and REIL [43]. Crucially, full-translation techniques use the expressivity of such powerful IRs to recover higher-level constructs such as control flow, basic blocks, and functions, that enables them to apply complex program-wide analyses and transformations.

Dynamic rewriters transform the program *during* program execution. This is achieved by using an instrumentation engine that inserts fine-grained hooks (PIN [76], DynamoRIO [28]) during native execution or by running the binary inside a virtual environment (QEMU [23]). Compared to static rewriters, dynamic approaches can perform much more precise and fine-grained modifications as they can observe control flow and progam state at runtime. However, modifications performed by such rewriters usually only persist for the duration of the execution run.

2.2 Binary Recompilation

Recompilation is a full-translation rewriting technique that ***lifts*** binary programs to a compiler IR with the goal of ***lowering*** it back down again, using the existing compiler infrastructure. The goal is to make available the rich analysis and transformation ecosystem of modern compilers, that is typically used with source-level programs, to modify binary programs. State-of-the-art recompilers, SecondWrite [14], BinRec [12], McSema [41], McSema [115], and Rev.Ng [40], target LLVM IR [69] due to its active community, modular design and the rich tooling support.

2.3 Semantics Recovery for Recompilation

Faithfully representing (lifting) a binary program in a higher-level representation, such as a compiler IR, requires the precise recovery of various program elements. This includes inferring the targets of indirect control transfers, marking function boundaries, identifying function prototypes (arguments and return values), locating program variables and assigning them types. This is hard as compilation is a lossy process; information about various source-level abstractions is unavailable in the binary artifact. In fact, the detranslation of computer programs is an undecidable problem [56].

Compilers can be instructed to emit metadata, such as debug information, symbol tables, and unwind tables for exceptions, that embeds useful information as part of the generated binary. Binary analysis tools such as BOLT [88] make use of relocation information that the linker emits for reordering functions. However, recovering precise information from binary metadata is a "best effort" implementation. For instance, optimizing transformations that aggressively inline functions, move variables from memory to register storage, or reuse stack locations for distinct variables within the context of a single function, do not make any guarantees about preserving precise debug information. BOLT also reports missing offsets for position independent code (PIC) jump tables, as they are removed by the linker. Moreover, as commercial off-the-shelf (COTS) binaries may be stripped of such metadata, recompilers cannot generally assume their presence.

Figure 2.1 provides an overview of the various sub-tasks that comprise the semantics recovery process for binary recompilation. We now discuss them in detail.

2.3.1 Instruction Recovery

The first step towards understanding an input binary is to identify and recover all executable machine code instructions that belong to it. Disassembly deals with decoding a stream of raw

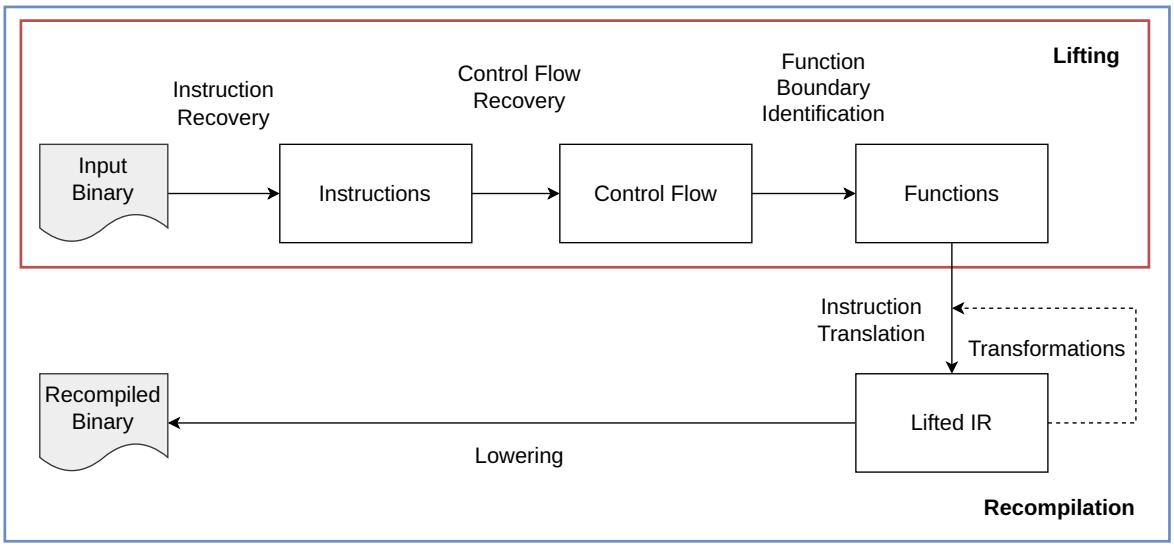


Figure 2.1: Overview of the various steps involved in the recompilation process. Dashed lines indicate optional steps.

bytes into individual instructions. This is hard, as compilers do not attach labels to distinguish between code and data references in the generated binary by default.

For instance, the compiler can embed non-executable data in the code section of the binary. Padding bytes may be inserted to align functions to page boundaries to improve instruction caching. Similarly, for source-level switch constructs, the compiler can generate jump tables and embed them inside the code section of the binary. It can also embed other read-only data in the code section, as observed in *literal pools* [6] in ARM binaries, to improve access distance to constant data.

On CISC style architectures such as x86/x64, variable length instructions can complicate instruction recovery. Due to the variable size of instructions, the same set of bytes can be decoded as distinct sets of instructions. Moreover, a specifically aligned data byte can lead to incorrect *local* disassembly, but realign itself with the correct disassemly stream at a later offset. As a result, a static error correction algorithm cannot reliably infer if correct disassembly was performed, as the two byte streams decode to perfectly valid but distinct x86 machine code. Consider the example shown in Figure 2.2. The same set of 5 bytes can be interpreted in 2 different ways depending

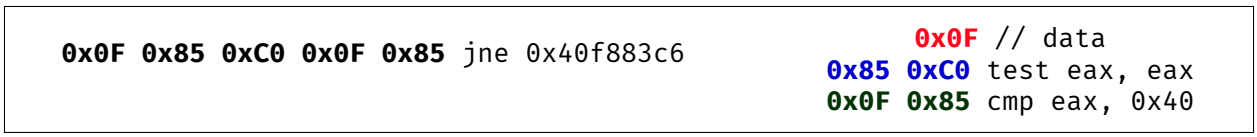


Figure 2.2: Different interpretations of the same set of bytes in x86 assembly.

on whether the first byte is considered a data byte (maybe inserted for alignment purposes) or a code byte. But, the two streams align back at the same code byte.

In practice, there are two static approaches to instruction recovery for binaries. Given a start address, ***linear sweep*** disassembles bytes as code until an illegal instruction is encountered. But, it does not differentiate between code and data bytes, which may lead to failures due to inline data being treated as code. For binaries that conform to well-known standards such as ELF [3] or PE [7], linear sweep can be applied to the entry point address that is parsed from the header.

However, the more commonly used approach is that of ***recursive descent***. Given a start address, the algorithm performs linear disassembly until a control transfer instruction is hit, at which point it retriggers the procedure at the discovered target. This way, the *worklist*-style algorithm recursively explores the control flow graph to find all reachable code from a given entry point. Disassemblers may retrieve an initial set of entry points by parsing function start addresses from the symbol table of the binary. State-of-the-art static recompilers either roll out their own version of a recursive descent style disassembly [14, 115] or use disassemblers that implement the same [2].

Note that recursive descent does not generate any false positives, but the procedure can miss entire basic blocks if control transfers to those are not correctly resolved. For instance, the basic block at address `0x40f883c6` in Figure 2.2 may not be discovered if the disassembly starts at offset 1 instead of 0. Therefore, incomplete discovery of executable code can leave out support for entire program paths in the recompiled binary. In practice, recursive descent can fail to discover all executable code due to the imprecise resolution of indirect jump and call transfers. Crucially, this reveals that the problems of control flow and instruction recovery in binaries are very much

```
void* secure_clear_memory(void* ptr, int
    size) {
    // Alternatives: memset_s,
    // memset_explicit
    memset(ptr, 0, size);
}
```

Listing 2.1: Call to memset inside a wrapper function.

```
secure_clear_memory:
  movsx   rdx, esi
  xor     esi, esi
  jmp     memset
```

Listing 2.2: Generated x64 assembly code with tail call to `memset@PLT`, gcc 14.1 `-O3`.

interleaved.

2.3.2 Control Flow Recovery

Control flow recovery is the process of identifying the points of control transfer between various instructions in the binary. Resolving control flow for instructions that encode their targets is trivial. However, *complete* control flow recovery is hard to achieve due to indirect jump and call transfers. Indirect jumps implement switch statements, such as jump tables, and position-independent code (PIC). They may also be used for tail calls, in the case when the function signatures for the caller match the callee. Modern compilers also implement support for tail calls to imported routines, such as in Listing 2.2. On the other hand, indirect calls implement virtual function calls in C++ and calls via function pointers in C.

Recovering the targets of indirect control transfers in binaries has been a topic of active research over the past decade [89, 111]. Static tools often use a combination of heuristics and program analyses, such as value-set analysis (VSA) [17], symbolic execution [101] or techniques like Simple Expression Tracking (SET) [40]. This enables them to predict an overapproximate set of targets that can be used to further explore reachable machine code. Disassemblers may also implement fine-grained heuristics that are tuned after observing numerous compiler generated code patterns for specific architectures. McSema [41] relies on IDA Pro [2] for recovering control flow information, but it performs poorly when recovering targets of indirect calls [89]. mctoll [115] employs heuristics to identify jump tables and resolve a subset of indirect intra-function control transfers.

It cannot identify possible targets of indirect calls precisely, either.

Multiverse [20] is a binary rewriter that implements the idea of superset disassembly, where they disassemble at every single offset in the executable code section of the binary. The various disassembly streams are then merged into a single control flow graph. Although this approach increases the generated code size, it is guaranteed to discover all possible code.

Completeness ensures that the recompiled binary supports *all* control flow paths that were a part of the input binary. On the other hand, *precise* control flow recovery is necessary to ensure that unintended paths are not added.

2.3.3 Static v/s Dynamic Approaches

The previous two subsections describe entirely static approaches to the problem of instruction and control flow recovery. Although fast, static approaches may suffer from incompleteness and imprecision due to the undecidability of underlying problem [56].

On the other hand, dynamic approaches observe concrete executions of the program to recover executable instructions and the control flow. They rely on techniques such as fuzzing using tools such as AFL [119], and AFL++ [45], to achieve high coverage and explore as many program paths as possible by providing different inputs [122]. Achieving high coverage for binary-only fuzzing (black-box) has been a topic of active research [79, 84, 117]. Concolic execution - which can be thought of as a hybrid between symbolic and concrete execution has also been effective in program exploration [32, 94, 103].

BinRec [12] is a dynamic recompiler that handles precise code and control flow recovery by design as it records information about paths realized during concrete execution runs of the input program. It uses S^2E [34], which uses QEMU [23] underneath, as a lifting frontend to trace binaries. S^2E implements a translation layer from QEMU's architecture-agnostic Tiny Code Genera-

tor (TCG) IR to LLVM IR to facilitate symbolic execution with KLEE [30]. BinRec inserts hooks that captures the translated LLVM IR along the paths that are exercised during tracing, and later merges the traces captured across different runs. As enabling deobfuscation of input programs is one of their goals, BinRec induces a tight coupling between CFG recovery and IR translation which imposes a significant tracing overhead. Each tracing run needs to setup the emulator-like execution environment (QEMU) which is inefficient. Overall lifting times are reported to be orders of magnitude costlier in comparison to a static recompiler [12].

With dynamic recompilers, the recovered binaries only reliably support the program paths that are observed during the tracing runs. Although this approach is sound, "you get what you see". However, a downside is that code paths that are not executed are not recovered - implying that the completeness of recompilation depends on the coverage of the input domain. BinRec uses a combination of high-coverage inputs and symbolic execution to achieve maximum coverage of the binary CFG.

2.3.4 Function Boundary Identification

Recovering functions comprises the problem of identifying function entry points, assigning basic blocks to them and marking their boundaries (in terms of instructions and basic blocks). The function abstraction is crucial in the context of recompilation, as it helps to scope program-wide transformations in the lifted IR. Rather than having to analyze the entire CFG of the program, off-the-shelf compiler optimizations benefit from analyzing individual functions in isolation. Also, if security critical vulnerabilities are to be patched in the input binary, it may make sense to replace entire function bodies with their safer implementations.

But, precise and complete function recovery is hard to perform statically. Functions can have multiple entry points to specialize for certain constant arguments [18]. This can misguide heuristics and pattern matching techniques that detect function starts by looking at prologue signatures.

```
extern int g(int a, int b) __attribute__
    ((cold));
int f(int *a, int *b)
{
  int result;
  if(*a > 0) {
      result = *a + *b;
  } else {
      result = g(10, *a) + 0xdeadbeef;
  }
   return result;
}
```

Listing 2.3: C code with function `g` annotated with attribute `cold`.

```
f(int*, int*):
        mov     eax, DWORD PTR [rdi]
        test    eax, eax
        jle     .L2 ; notice the jump
        add     eax, DWORD PTR [rsi]
        ret
f(int*, int*) [clone .cold]:
.L2:
        push    rdx
        mov     esi, eax
        mov     edi, 10
        call    g(int, int)
        pop     rcx
        sub     eax, 559038737
        ret
```

Listing 2.4: Generated x64 assembly code (`-O2`, gcc 11.1), note the `jle` and the cold clone of `f`.

Entire function bodies can be inlined in other functions, for reducing call overheads or to expose other optimization opportunities.

After correctly identifying function starts, associating basic blocks to functions can be difficult due to non-contiguous functions and code-sharing [83]. For instance, functions may have code gaps due to inline data or jump tables or they may share code with other functions for optimizations reasons.

Tail calls / jumps make it hard to determine function boundaries since they end in a jump or call-based control flow transfer instruction rather than a return. As a result, function epilogues that usually consist of a local stack cleanup (e.g. `add rsp, 0x80; ret`) may be missing for some functions. Moreover, functions may be non-returning, and calls to library functions like `exit()` mark boundary points. In such cases, tools that use known epilogue patterns to identify function boundaries may incorrectly disassemble beyond the call / jump.

An interesting problem deals with the handling of split functions in binaries compiled with modern GCC and LLVM. The compiler may *split* function bodies as an optimization to improve inlining [48] decisions based on information collected through profiling or if provided by the user

through function attributes (such as `cold`). Consider the examples in Listings 2.3 and Listing 2.4. As the user indicated that the function `g` is `cold`, the compiler generates a clone of the function `f` that contains the slow path - condition `*a <= 0` and the call to `g`. Note that a new symbol in the symbol table for the function `f [clone .cold]` is created. But unlike usual function calls, the cold clone is reached through a direct jump (`jle`) instruction rather than a `call` instruction.

Another special case is that of callback functions (e.g. comparison function passed as an argument `qsort()`), that act as external entry points. It is likely that (direct or indirect) calls to such functions are not observed in the binary as their pointers are passed externally. However, such functions typically contain a prologue for preserving saved registers and ensuring the stack setup, making them easier to be identifiable by a disassembler.

A straightforward approach to identifying function starts is to parse debug information such as DWARF [36] or symbol table information found in the `.symtab` and `.dynsym` sections. In practice, static tools such as IDA Pro [2], BinaryNinja [109] and Ghidra [85] also use a combination of pattern matching and heuristics that are derived after carefully observing code patterns generated by compilers such as GCC, LLVM and MSVC [89, 111]. Moreover, call targets - both direct and indirect - that are discovered during the instruction recovery process can be used to identify a subset of function entry points.

NUCLEUS [15] details a compiler-agnostic approach that uses a global call graph, along with a weakly connected component analysis to identify function bodies based on call-and-return edges. Dynamic approaches may need to rely on such an algorithm to recover functions using run time information such as call-and-return edges. BinRec does not implement any such technique due to which the entire trace of instructions, obtained after merging, is represented as a single function in the lifted IR. This impedes compiler optimizations and prevents the user from making surgical modifications to the IR.

Note that, recovering functions also reveals information about the stack layout of the program.

This can be useful to *lift* objects from the emulated stack of the program to the native stack, leading to better off-the-shelf compiler optimizations [44]. Intra-function memory loads and stores that target the stack can be used for variable detection in binaries [71, 72, 102].

2.4 IR-level Representation Of Binaries

Faithfully representing the semantics of individual hardware instructions in a compiler IR such as LLVM IR is challenging. This is because there is a fundamental difference in the abstractions available at the hardware level and at the IR level. For example, the instruction `add eax, dword [ecx]` on x86:

- Loads a 32 bit integer stored at the memory address in register ecx.
- Adds it to integer value stored in eax.
- Stores the result back in eax.
- (Side effect) Updates the EFLAGS register based on the result of the operation.
- (Side effect) Advances the PC forward by 4.

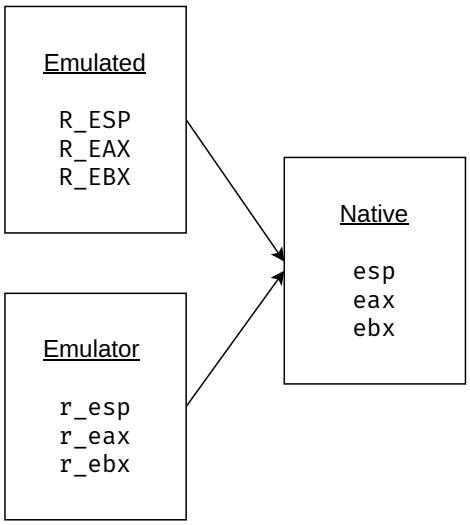
There are multiple points of differences when translating such an instruction to LLVM IR. For instance, LLVM IR defines explicit instructions for load/store/read-modify-write (RMW) memory operations. So, combining the load from memory along with an addition operation is not possible. Assuming that the add and the load are broken into two distinct operations, we encounter further challenges when trying to precisely model the side effects - such as updates the to `EFLAGS` and `PC` registers.

```
%1 = load ptr, ptr @ecx ; load from global ecx
%2 = load i32, ptr %1 ; load from memory
%3 = load i32, ptr @eax ; load from global eax
%4 = add i32 %2, %3 ; perform add
store i32 %4, ptr @eax ; store result back to global eax
%5 = call i32 helper_compute_eflags(i32 %4) ; compute eflags
store i32 %5, ptr @eflags ; update eflags
store i32 4224509, ptr @pc ; store address of next instruction
```

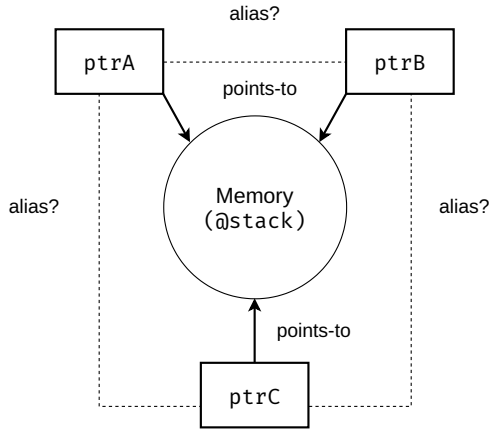
Listing 2.5: Lifted LLVM IR for `add eax, dword [ecx]`

To solve this, recompilers typically implement lifting by performing line-by-line translation of machine code to semantically equivalent LLVM IR. The lifted IR aims to *emulate* the execution of each hardware instruction on a virtual CPU state that consists of registers, flags, and stack memory. This idea is similar to a virtual execution environment such as QEMU, that acts as an emulator for native binaries. Listing 2.5 shows the lifted IR for this instruction. However, note that the IR generated this way is highly verbose and unrefined, which is a central cause of performance slowdown of the recompiled binary.

Emulated Registers. As a byproduct of the emulation, the recompiled binary has two sets of registers in the IR. The "emulated" set, that belong to the virtual CPU state, models the registers in the input binary. But, the "emulator" also requires access to registers for passing arguments to functions and performing function-local computations. This leads to the use of more (virtual) registers as compared to the input binary (ref. Figure 2.3a). However, on compilation, the two sets compete for the same hardware registers creating additional register pressure and increasing the number of spills – negatively impacting recompiled binary performance. An exception case is when the binary is being recompiled for a different architecture, one with more general purpose registers (x64 -> AArch64), but that still does not preclude the performance overheads due to other emulation effects.



(a) Emulator and Emulated registers in the IR contesting for the same set of hardware registers.



(b) Compiler failing to establish aliasing relationships between distinct stack pointers.

Emulated Stack. The stack is a critical component of the execution model of binary programs as it is used for storing local variables and register spills. In the emulator model, the recompiled binary also works with two stacks - an "emulated" stack and the "native" stack. The emulated stack models the call stack of the input binary by storing its local variables, spills, function arguments and return values. On the other hand, the native stack is used for storing local variables, spills, function arguments and return values of the emulator itself.

Typically, recompilers allocate a large byte-array that represents the emulated stack. Before jumping into recompiled binary execution, instrumentation is inserted to copy over command line arguments and initialize the emulated stack pointer register to point into this allocation. But, representing the stack as a global array of bytes negatively impacts memory usage and performance. The fundamental reason for this is that the allocated stack storage (`@stack`) remains opaque to the compiler as all stack-based loads and stores target the same memory object (ref. Figure 2.3b). This prevents precise alias analysis and the formation of use-def chains (even within function boundaries) which impedes downstream off-the-shelf compiler optimizations that rely

on this information.

2.4.1 Code and Data Reference Relocation

Relocation is a linker concept for ensuring that program elements defined in different source files correctly refer to each other after linking. As recompilation imposes a new layout on the binary, sound program-wide transformations on the lifted IR require that ***all*** code and data pointers be relocatable. Failure to do so may lead to runtime crashes, due to the transfer of control flow to an invalid code address or the derefence of an invalid data pointer (e.g. access to a non-relocated global variable).

```
0x40d67c: mov    %rsp,%rsi
0x40d67f: mov    0x0,%edx
0x40d684: call   0x421efb ; direct call, need to relocate
0x40d689: mov    %rax,%rdx
0x40d68c: mov    0xffffffffffffffff,%rax
0x40d693: test   %rdx,%rdx
0x40d696: jne    0x40d65e ; direct jump, need to relocate
0x40d698: mov    0x10(%rbx),%rax
0x40d69c: mov    0x8(%rax),%rdi
0x40d6a0: mov    0x0,%esi
0x40d6a5: call   *%rcx ; indirect call, difficult to relocate
```

Listing 2.6: x64 assembly code snippet showcasing the necessity to relocate absolute addresses for control transfer instructions.

For instance, consider the assembly snippet in Listing 2.6. The recompiler has to symbolize the absolute addresses used as operands for jump (`jne`) and call (`call`) instructions, such that they can be retargeted to their new addresses after the layout changes. This is trivial to achieve for direct references. But indirect references could be materialized in registers or memory at runtime, which imposes the requirement to perform precise pointer tracking. The `call *%rcx` instruction

in Listing 2.6 is such an example.

Crucially, differentiating between integers and pointers is hard at the binary level. Most programming language standards do not mandate any specific hardware level representation for pointers. As a result, there is no inherent bit pattern or binary-level metadata that distinguishes between them. Binary analysis tools often implement a combination of forwards and backwards data flow analysis passes for propagating pointer information [111]. For instance, we can infer that a register contains a pointer value at its site of use - e.g. if it is used to dereference memory, or passed as a pointer argument to a known library function. This information can then be propagated backwards through the CFG. On the other hand, forward passes usually track the data flow of pointers from their definition site to the various uses. Common examples of *def* sites are returns from library calls such as `malloc` or instructions that perform memory addressing computations (e.g. `lea` in x86/x64) using known pointer values, such as the stack pointer.

Code pointers. Relocation of all code pointers requires precise and complete control flow recovery. This way, the recompiler can symbolize constant values by identifying if they point to known basic block or function start addresses. A simple way of resolving this is to use a pre-computed lookup table that acts as a trampoline, redirecting control flow by mapping old addresses to new (relocated) ones [51].

A special case is that of callback functions. To ensure correct support for external entry points, recompilers need to precisely identify and rewrite *all* instances of function pointers passed to external procedures. Rev.Ng supports external entry points through static linking of libraries and treating them as indirect calls. However, analyzing and rewriting statically-linked code is generally infeasible, as it incurs a substantial performance overhead and does not scale. McSema and mctoll try to statically identify function pointer arguments passed to external functions to rewrite them. But, tracking pointer values in machine code, especially if they are passed across function boundaries, can be hard. Also, it may be impossible to precisely solve this problem statically if pointer values are materialized in registers or memory during execution.

Data pointers. Symbolizing data pointers requires precisely identifying and rewriting all references global variables in the binary. Similar to approaches that resolve indirect control transfers, recompilers use a combination of heuristics and value-set analyses to identify if a register or memory location contains a data pointer. A common heuristic is to track the propagation of constant values that lie in the address range of the data section of the binary.

But, incorrect symbolization of a constant value to a pointer may lead to unsound behavior. For instance, the recompiled binary semantics may completely change if a constant value is incorrectly inferred to be a code pointer, and relocated after lowering. This has been shown to be the case with McSema due to its use of IDA Pro's heuristics for symbolization [12].

An alternative method to handle relocations, used by BinRec, is to map the entire input binary at its original load address as part of the recompiled binary image [12]. BinRec does not attempt to identify and rewrite function pointers passed to libraries, but instead inserts trampolines at the original addresses of all external entry points that it traces. 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. This approach also eliminates the need to rewrite global data references as data objects in the input binary are retained at their original addresses. But, a major drawback of this approach is the memory overhead of mapping the entire input binary in the address space of the recovered binary.

Rewriters may use the relocation metadata available in modern position-independent (PIE) binaries. For instance, RetroWrite [42] and Egalito [112] use the insight that for x64 PIE binaries, (1) all absolute pointers have relocations, (2) the compiler typically uses relative pointers in the data section for jump tables and, (3) program counter (PC)-relative pointers in code sections are easily relocatable. However, a central target for rewriting applications are legacy binaries, that may not contain such metadata.

2.4.2 Lifting v/s Recompilation

For the rest of the dissertation, we establish a difference between the terms ***lifting*** and ***recompilation***. Lifting defines the process of translating machine code to a compiler IR (we focus on LLVM IR). This includes the sub-tasks of instruction recovery, control flow recovery, function boundary detection and individual instruction translation to semantically equivalent LLVM IR.

On top of lifting, recompilation involves implementing support for the entire round trip that generates a new binary with the semantics and behaviors of the original binary. After lifting to IR, recompilers typically,

- link in support for precisely emulating the virtual CPU state, e.g. adding stack memory, modeling registers and flags, and defining a new entry point.
- perform *semantics preserving* transformations and optimizations, such as devirtualizing the lifted IR to improve performance.
- handle code and data pointer relocations.
- lower the IR to generate a new binary.

Tools like Rev.Ng [40], and RetDec [67] exclusively classify themselves as lifters rather than recompilers. This implies a fundamental difference in the downstream tasks that they concern with. For instance, a central goal of RetDec is to lift binaries to LLVM IR for the purpose of decompilation (to C-like pseudo code). As a result, they can afford to use unsound heuristics, such as to recover program variables, to refine their IR since the decompiled code is not intended to be recompiled back down. Recent work uses debug information to refine the lifted IR, but they do not target recompilation [120].

Chapter 3

Polynima - Practical Hybrid Recompilation for Multithreaded Binaries

Although recompilers aim to bridge the divide between the modern compiler ecosystem and binary programs, the current state-of-the-art tools do not deliver on this promise. Crucially, they fail to generally handle the various challenges posed by multithreaded programs that are ubiquitous in the current software space.

To address these challenges, this chapter introduces Polynima, a binary recompiler that reliably supports x86/x64 multithreaded binaries while introducing a moderate 1.07x slowdown. We propose a hybrid control flow recovery approach that combines the benefits of static and dynamic techniques while providing an efficient strategy to handle unknown paths. Polynima enables the use of the rich LLVM compiler ecosystem to fix and improve legacy multithreaded binaries, which we demonstrate by mitigating a critical synchronization issue in an FTP server binary. We also leverage the functional IR to introduce a novel dynamic analysis to detect implicit synchroniza-

tion primitives in binaries, which we use to further improve performance of the output. Finally, we evaluate the *generality and correctness* of Polynima by recompiling a diverse set of *real-world, multithreaded* binaries and benchmark suites. To our knowledge, Polynima is the first recompiler to be able to do so.

Polynima improves over the state-of-the-art by combining sound static lifting with the ability to perform optional, instrumentation-based dynamic analyses that refine the lifted IR.

3.1 Current Limitations

We first discuss the specific challenges that recompilers have to face pertaining to multithreaded binaries and the limitations in the state-of-the-art.

3.1.1 Control Flow Recovery

Section 2.3.2 and 2.3.3 provide an overview of the approaches to control flow recovery in current recompilers. Static recompilers are fast (to lift) but employ heuristics to predict targets of indirect control transfers which can be imprecise [89]. Dynamic recompilers [12] provide precision but are highly inefficient due to the tracing overheads and suffer from the problem of coverage. It is evident that approaching the problem using an *entirely* dynamic or static manner is not ideal.

Moreover, static and dynamic approaches to control flow recovery both suffer from the problem of completeness. A control flow miss occurs when, during its execution, the recompiled binary attempts to perform a control flow transfer to an address that was not discovered during lifting. Such misses are frequently triggered by non-deterministic program behaviors [12] that may arise due to explicit (`getenv()`, `rand()`) or implicit (using pointers as hashmap keys) sources of randomness. Crucially, due to the various thread interleavings that are possible at runtime, non-

deterministic behaviors are particularly common in multithreaded binaries. None of the existing static recompilers implement support for such control flow misses.

A sound way of handling control flow misses is to terminate binary execution at that program point, as it does not introduce any divergent behaviors. However, this strategy needs to explicitly encoded as part of the IR as the compiler may optimize certain paths and introduce semantics that were not a part of the original binary. For instance, if a single target to an indirect call is known, the compiler may inline the function call and execute the inlined path for all possible inputs.

BinRec implements *incremental lifting* to handle this issue. When a control flow miss occurs, BinRec terminates execution of the running program, records the virtual PC value and initiates a new trace of the original binary using their lifting frontend. To keep the tracing overhead manageable, this trace is started at the newly discovered address instead of the beginning of the program. Since the incremental trace is started at an arbitrary address, it is prone to trigger runtime faults before discovering any new targets due to uninitialized stack and heap memory. To mitigate this, BinRec performs path exploration only until the next conditional control transfer which is inefficient.

Also, one of the core promises of recompilation is cross-ISA (instruction set architecture) translation, which is valuable for programs compiled for legacy architectures. But, current dynamic approaches rely on access to the original execution environment or an emulator to recover the CFG, which is not guaranteed. Consider the recompilation of a binary originally compiled for a legacy ISA. Replicating the intended execution environment for a such a binary may not be possible, as it may depend on a specific version of the operating system, system variables, etc. However, it may be feasible to write a static translator that performs line-by-line lifting of machine code to LLVM IR. This way, once a recompiled binary that targets a more modern processor architecture is obtained, it can be experimented with. This case of *on-device* lifting, additionally motivates the use of a combination of static and dynamic techniques for efficient and precise

binary recompilation.

3.1.2 Hardware Atomic Instructions

Multithreaded binaries leverage program constructs, such as those provided by the programming language, compiler, or the underlying hardware, that pose new challenges for recompilation. Atomic instructions are critical for implementing synchronization primitives, such as locks and semaphores, in machine code. On the x86/x64 hardware, this includes read-modify-write (RMW) operations (e.g., `lock add`, `lock inc`) and compare-exchange operations (e.g., `lock cmpxchg`). Atomic memory accesses on the x86/x64 architecture assert that all observers see the access as having happened or not happened at all, and never partially happened [58]. The executing thread has exclusive ownership of the data for the duration of the instruction to ensure that no partial state is ever exposed to other observers on the system. Therefore, precisely mapping these hardware instruction semantics to LLVM IR is difficult.

Consider the translation of the `lock cmpxchg dword ptr [rsi], ecx` instruction as a representative example. 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, else 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. Note that all of the sub-operations are executed as part of the same hardware instruction. However, the programming model and the set of available abstractions in lifted LLVM IR differ notably from that assumed by the underlying hardware. For instance, it is not possible to represent the update to the (virtual) `eax` register and the atomic compare-exchange operation as part of an indivisible IR instruction.

Of all the recompilers, only McSema supports the *lifting* (binary to IR) of hardware atomic instructions to LLVM IR by using the appropriate compiler intrinsics. Unfortunately, its authors

conveyed to us that its recompilation capabilities (binary to IR to binary) are experimental and need an expert operator to fix issues manually.

3.1.3 Handling the Program Stack

Stack memory is critical to program execution as it stores function-local variables, spilled register values, and arguments for calls. Recompiled programs usually work with two stacks of execution, (1) the *native* stack, that contains variables and spills that are a byproduct of the emulation, (2) the *emulated* stack, that includes variables and spills of the input program. Recompilers such as McSema, BinRec, and Rev.Ng model the emulated stack as a global array of bytes. However, their implementation is not general as they do not handle the multithreaded case, where each thread of execution needs to work with its own emulated stack.

Some recompilers split the emulated stack into individual chunks and move them to the lifted program's native stack to mitigate the resulting performance slowdowns. For instance, mctoll performs static analysis to identify the maximum bound on the per-function stack frame size and creates a (function-) local allocation that represents the original program's stack frame. But, this approach is not general as, we later show in Chapter 4, statically inferring the maximum frame size of functions in binaries is hard. In fact, the frame may not be bounded at all for programs that call `alloca` (or one of its many variants) with a dynamically determined size argument or use Variable Length Arrays (VLAs). Insufficiently allocated stack frames could lead to runtime faults after recompilation due to out-of-bounds frame local accesses reaching into unknown memory.

Moreover, this optimization of recovering the per-function stack frame relies on precisely identifying and translating *all* accesses to the local stack frame in the original program. Statically performing this procedure is largely heuristics-driven, as identifying if any stack reference escapes is undecidable in the general case. This issue is critical for lifting multithreaded binaries as imprecision in identifying stack-exclusive accesses may lead to erroneous and unsynchronized

shared memory writes. Due to the lack of generality of this approach, previous work [98] that builds on mctoll could not evaluate specific binaries from the Phoenix benchmark suite [97].

3.1.4 Callback Functions

Correct handling of callback functions is necessary 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 to be *external* entry points, as the control flows in to the binary program from external library code.

We discussed the various approaches to handling callback functions in Section 2.4.1. Of the static recompilers, only Rev.Ng implements a sound approach to handling callback functions. But, their tool is only applicable to binaries that are statically linked which impedes its practical application. Such binaries are uncommon in off-the-shelf software, as they create a large disk and memory footprint. BinRec's approach is sound but it does not handle the case when the entry point may be executing as part of a different thread. Specifically, it does not correctly initialize the virtual CPU state and the thread-local emulated program stack on entry which may cause faults at run time.

3.1.5 Memory Access Reordering

Information about the relative ordering of memory accesses, which may be specified implicitly using synchronization barriers or explicitly using source annotations, is lost during program compilation. Hence, the lifted IR obtained from such binaries contains no ordering information. Failure to preserve the original program ordering may lead to erroneous and divergent program outcomes due to the compiler reordering shared memory accesses at the IR-level.

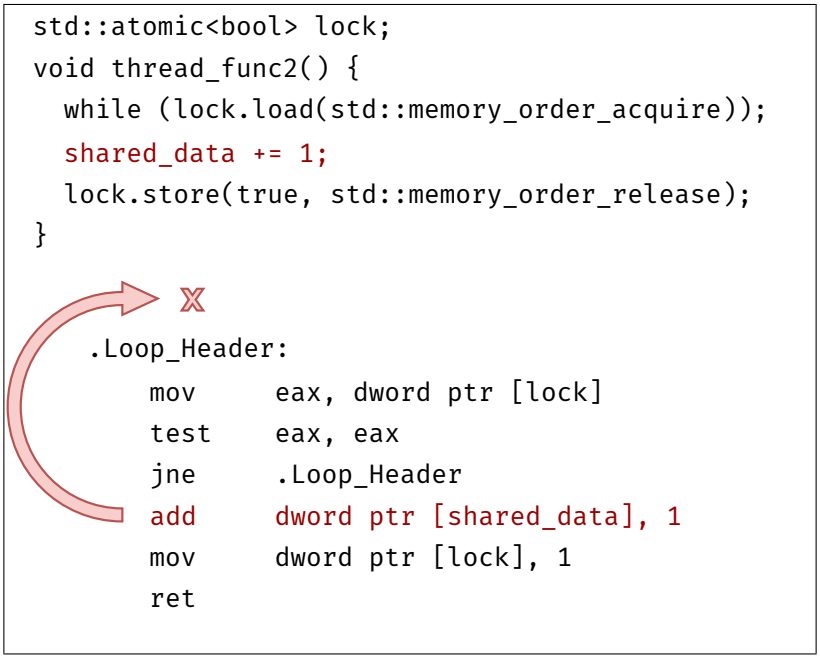


Figure 3.1: Memory access reordering at the IR-level may lead to erroneous outcomes. The write to `shared_data` may be reordered across the critical section during recompilation due to the lack of original ordering semantics.

Consider the example shown in Figure 3.1 which represents a shared memory access that is synchronized using a spinlock. The source-level memory ordering semantics attached to the *acquire* load and the *release* store of `lock` assert that write to `shared_data` will not be reordered across the critical section. However, the generated machine code implicitly encodes these semantics due to the lossy nature of the compilation process and the guarantees provided by the underlying x86/x64 ISA. For instance, (1) as naturally aligned stores and loads upto 64 bits are guaranteed to be atomic, the compiler emits an ordinary `mov` for the load operation, (2) the strong Total Store Ordering (TSO) model (with store forwarding) prevents the Store-Store reordering between the `add` and the `mov`, But, after lifting, the compiler is free to reorder these memory accesses in the IR (such as in the case of `shared_data`) which may break original program semantics.

To remedy this issue, Lasagne [98] formalizes the idea of the LLVM IR Concurrency Model (LIMM) and discusses a sound strategy to lift memory accesses in multithreaded binaries to LLVM IR. They insert appropriate fences for each memory access preventing the compiler from reordering

them. As fences can be costly for performance and hinder off-the-shelf optimizations, they propose certain optimizations - (1) merging adjacent fences that induce redundancy, (2) removing fences for stack-exclusive accesses. Note that, the second optimization is not always sound as it requires an analysis procedure that proves that the stack object does not escape the thread-local scope.

Recent work [22] has shown that Lasagne's fence placement strategy may impose stricter restrictions than necessary for specific programs, incurring a high performance cost for recompiled binaries. Lasagne primarily targets cross-ISA translation to ARM (a weaker memory model), which requires that the IR impose the strict x86/x64 memory model for all memory accesses, except those that target the thread-local stack. But, when recompiling binaries for the same architecture, almost all inserted fences are superfluous for programs that synchronize shared memory accesses through exclusive use of externally provided barriers and primitives (e.g., those provided by the *pthread* library). This occurrence is common, as correctly implementing custom primitives is hard and programmers often rely on third-party libraries to achieve this. Also, if the programmer can reliably partition the data such that the different threads access mutually exclusive elements, explicitly sychronizing accesses to the shared memory object is unnecessary. In fact, most programs in the Phoenix benchmark suite exhibit a combination of these properties [97].

3.2 Design and Implementation

Polynima is a full-transformation recompiler consisting of modules that perform control flow recovery, translation of machine code to LLVM IR, optimization and lowering. Crucially, recompiled output generated through static-only analyses acts as a functional 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

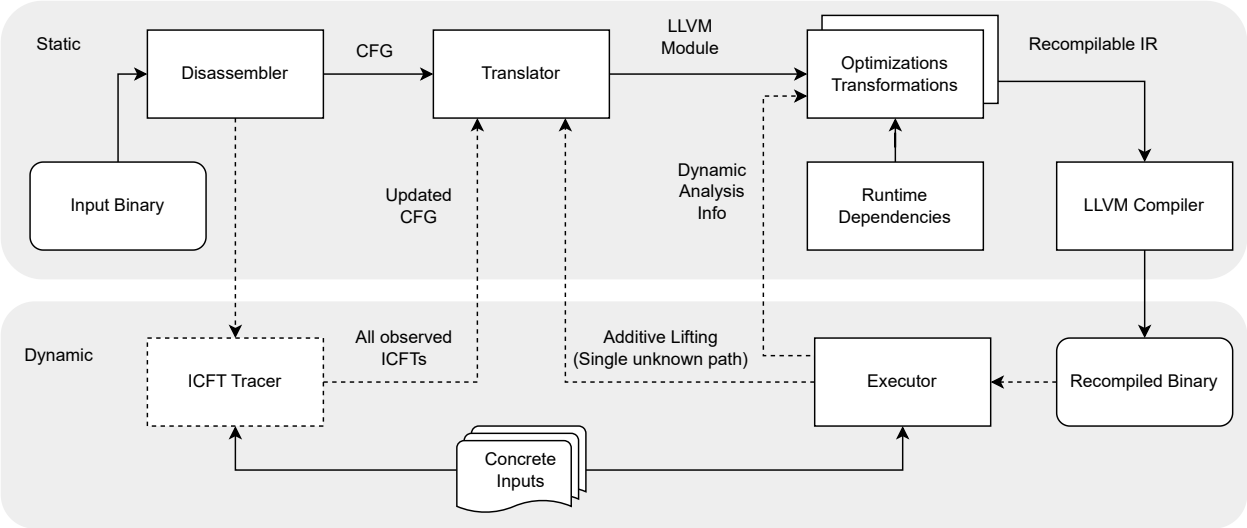


Figure 3.2: Overview of Polynima. Dashed lines indicate optional steps.

of this representation. Figure 3.2 gives an overview of the system architecture.

3.2.1 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. We assume that the program stack grows downwards and that the stack pointer register (`esp/rsp`) points to the top of the thread-local stack.

Stack Switching. 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.

Consider the example of the `fwrite` call shown in Figure 3.3. The `fwrite` function is part of the libc library that is dynamically linked to the binary. In the generated assembly, the call to `fwrite` is made through the Procedure Linkage Table (PLT), with the arguments being passed in registers

```
size_t result = fwrite("Simple thread example\n",
                       1, 22, stdout);



.LC1:
        .string "Simple thread example\n"

        mov     rax, QWORD PTR stdout@GOTPCREL[rip]
        mov     rcx, QWORD PTR [rax]
        mov     edx, 22
        mov     esi, 1
        lea     rdi, .LC1[rip]
        call    fwrite@PLT
```

Figure 3.3: `fwrite` call source and the associated assembly code.

per the System V ABI [80] for x64 binaries.

```
void external_call_trampoline(void* call_addr) {
    asm(
      // Translate emulated state to native
      mov    @rdi, %rdi
      mov    @rsi, %rsi
      // ....
      // Save native rsp in a call-preserved register
      mov    %rsp, %rbx
      // Switch stack
      // Native rsp points to emulated rsp for the duration
      // of the call
      mov    @rsp, %rsp
      // Make function call
      call   *call_addr
      // Switch stack back
      mov    %rbx, %rsp
      // Translate output registers to emulated state
      mov    %rax, @rax
      mov    %rdx, @rdx
    );
}
```

Listing 3.1: Helper function that implements stack switching.

The core idea of stack switching, as shown in Listing 3.1, is to translate the emulated state into native for the duration of the call. For calls with unknown interfaces, recompilers have to make

```
; Moves into emulated registers
; ...
%1   = load ptr, ptr @fwrite
%ret = call i64 external_call_trampoline
    (%1)
```

Listing 3.2: External call with stack-switching.

```
; Moves into emulated registers
; ...
%1 = load ptr, ptr @rdi ; void *ptr
%2 = load i64, ptr @rsi ; size_t size
%3 = load i64, ptr @rdx ; size_t nmemb
%4 = load ptr, ptr @rcx ; FILE *stream
%ret = call i64 @fwrite(%1, %2, %3, %4)
```

Listing 3.3: External call for functions with known signatures.

conservative assumptions about the function such as - (1) it may read from all input registers, (2) it may read/write to the program stack, (3) it may write to all output registers. As a result, the helper function translates all emulated registers to native and then forces the native stack pointer to point to the top of the emulated stack.

Listing 3.2 showcases the call to `fwrite` in the IR where the function signature is unknown. We create an external symbol for the unknown function, retrieve its address and pass it to the helper. On the other hand, Listing 3.3 shows how Polynima handles external calls with known interfaces. Depending on the number and type of the arguments, we query the ABI of the input binary to extract the specific argument values from the virtual CPU state. Then, we replace the call to the helper with a direct call to the external function as shown in Listing 3.3.

Note that, in the case that 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 in the context of the native stack.

Polynima operates on inputs without relocation information. This is typical of most legacy binaries which are our primary application targets. To handle code and data pointer relocations, we map the input binary at its original load address as part of the output. Therefore, the output contains the original binary code in addition to the recompiled lifted code.

We do not support threading mechanisms that can be achieved through `get/setcontext()`, `make/swapcontext()`, and `long/setjmp()`. Binaries may use threading models and synchronization primitives as exposed by POSIX threads (`pthreads`), C++11 standard (`std::thread`), C11 standard (`thread.h`), 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 builtins, such as the `__sync_` variants, that typically lower to use hardware atomic instructions.

We do not handle lifting of the `syscall` instruction. This occurrence is rare, as portable software usually relies on the existence of native shared libraries (such as `glibc`) on the target system to interact with the kernel. We currently do not support binaries with self-modifying code. Additive lifting enables us to recompile binaries with overlapping instructions and obfuscated control flow by design, but we do not evaluate our prototype on that capability barring a hand-written example. As emulation-based lifting often lifts individual hardware instructions into multiple LLVM IR instructions, it is difficult to ensure that intermediate virtual CPU state is not visible to other observers. Therefore, we assume that the underlying memory model does not imply the support of precise exceptions as it requires that the recompiled binary preserve semantics for instruction rollback when interrupted by the CPU.

3.2.2 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 / virtual program counter (PC). Each switch case represents a possible value that the PC could assume in the original program, and is mapped to the corresponding lifted block in the IR. An example is shown in Listing 3.4 that demonstrates the translation for the instruction `call r14`.

```
; 0x00405bfd  mov rsi, r13
; 0x00405c00  mov edi, 0x30
; 0x00405c05  call r14
; ----------------------
BB_405BFD:
; ...
store i64 %r14, i64* %pc
; Switch based on known indirect call targets
switch i64 %pc, label %error [
  i64 4213776, label %BB_CALL_indirect1391
]
; ----------------------
; BB that implements call to known target of the indirect call
BB_CALL_indirect1391:                ; preds = %BB_405BFD
; ...
%474 = call i8* @Func_default_logger(i8* %473)
; ----------------------
; Additive lifting
; Default case jumps to routine that records new call target
error:                               ; preds = %BB_405BFD
  %1688 = load i64, i64* %pc
  call void @binrecrt_record_pc(i64 4217853, i64 %1688)
  unreachable
; ----------------------
```

Listing 3.4: Lifted LLVM IR for `call r14`

But, as obtaining a set of possible targets for an indirect control transfer is a hard problem, Polynima implements a hybrid approach that can use static as well as 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 *uses but does not expect* disassembler-provided targets for indirect jumps and calls, benefitting from advances in static CFG recovery [62]. As the control flow is

conditioned on the actual PC value seen 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 instrumenting the terminating switch statements of all indirect blocks to jump to a custom runtime after encountering an unknown PC value. On encountering a new path, the runtime updates the on-disk representation of the CFG with this information and then stops program execution. In Listing 3.4, the `%error` block implements call to the custom runtime that records newly observed targets.

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 to static CFG recovery techniques for handling control flow misses.

Crucially, additive lifting enables *on-device* lifting. Users can statically generate a fully functional recompiled output, that supports known control transfers, for their target environment through Polynima. This is possible as the recompilation process enables the linking of new libraries, patching unsupported instructions and compiling for a different ISA. With the newly gained ability to natively run the program on the target architecture, unknown paths can be additively recovered during program execution.

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. In fact, such behaviors are particularly common in multithreaded machine code due to the various thread interleavings that are possible at runtime. We also observed this in binaries from the SPEC benchmark suite where pointers were being used as keys into hashmaps.

3.2.3 Hardware Atomic Instructions

Support for hardware atomic instructions is necessary to generally handle multithreaded machine code. A naive approach to their translation is to decompose them into distinct loads and stores, with all the accesses synchronized using a global (spin)lock. This maintains the guarantees of exclusive access to memory and other ordering constraints. But, a major drawback is that *all threads* executing an atomic instruction, irrespective of whether the referenced memory locations alias, have to unnecessarily (spin)wait.

To optimize this, we map atomic instructions to the appropriate compiler builtins at the LLVM IR-level during lifting. Listings 3.5 and 3.6 show the translated IR blocks for both the approaches.

```
lock(@global_lock)
%temp = *%rsi
if %eax == %temp:
  %flags.z = 1
  *%rsi    = %ecx
else:
  %flags.z = 0
  %eax     = %temp
  *%rsi    = %temp
unlock(@global_lock)
```

Listing 3.5: Naive.

```
compiler_barrier()
%old = %eax
%new = %ecx
%orig = cmpxchg *%rsi,
        %old, %new seq_cst
%flags.z = %orig == %old
if !%flags.z:
  %eax = %orig
compiler_barrier()
```

Listing 3.6: Optimized.

Here, we perform the write to the (virtual) `eax` as part of a separate instruction that depends on the result of the `cmpxchg`. However, we need to ensure that (1) the loads from the virtual registers (`eax, ecx`) are not reordered after the `cmpxchg` and before any other stores that target them. (2) the conditional store to `eax` is not reordered after any use of `cmpxchg`. To prevent such instruction reorderings, we mark the `cmpxchg` as sequentially consistent (`seq_cst`) and surround the translated IR block with compiler barriers. Since registers are not accessed indirectly, we can be certain that no other thread will race to write to the storage location of the `eax` register. We manually check the correctness of such translations for all supported hardware atomic instructions.

To preserve atomicity guarantees for memory operations asserted by the ISA, we maintain original alignments for, (1) global variables, by placing them at their original addresses, (2) program stack, by initializing the emulated program stack with the ISA mandated alignment.

3.2.4 Per-thread Stack and Callbacks

Polynima-lifted IR operates on a virtual CPU state that consists of 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 [44]. Functions take as arguments output registers (registers they may read and write to) and input registers (regis-

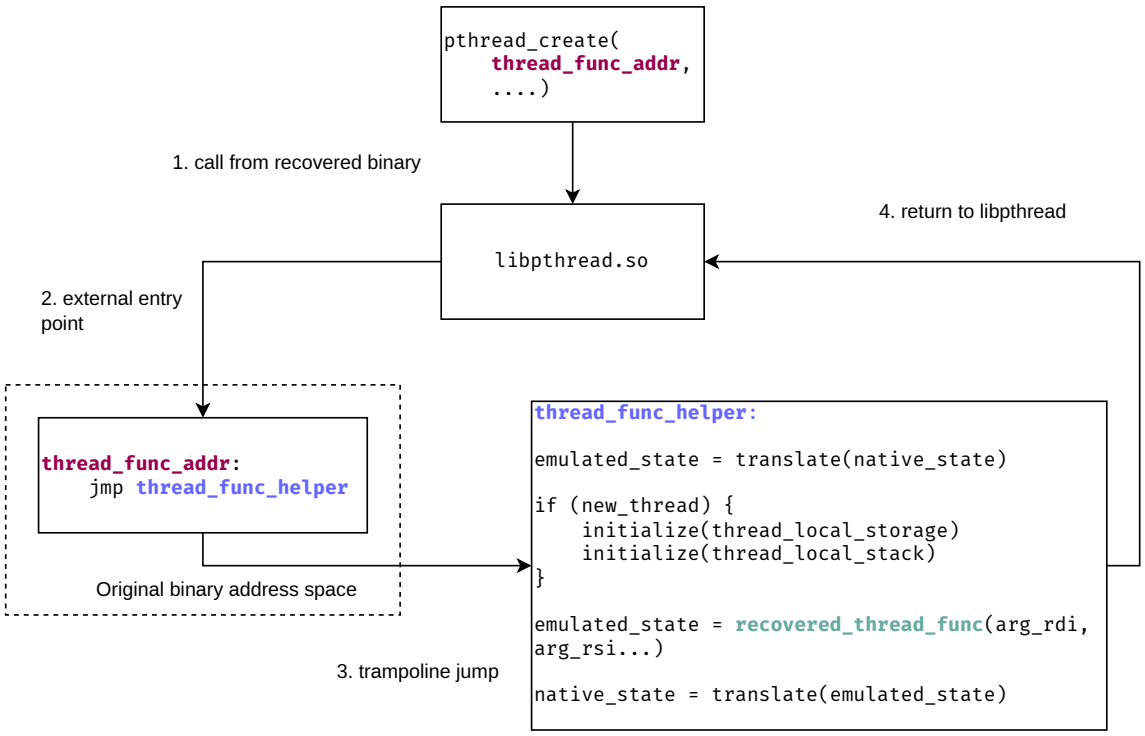


Figure 3.4: Execution flow on external entry into the recovered binary address space.

ters they 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` ensuring that each thread operates on its own copy of the virtual state. Figure 3.4 shows an overview of the callback handling process.

External library calls take function pointers as arguments when, (1) performing callbacks, such as in the case of `qsort` that requires a user-defined comparator function, (2) spawning new threads of execution, such as in the case of `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 also 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.

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. For callbacks executing in the same thread, the recovered function observes a different region of memory than the emulated stack. Specifically, it operates on the locally allocated byte array, as opposed to the emulated stack that is passed to the recovered `main` function. However, the library code and the native function do not assume any implementation-specific details about the memory model and only abide by the contract specified by the ABI. Hence, original program semantics are maintained irrespective of whether the callback is executing in a different or the same thread.

Callback Wrapper Optimization. To achieve sound static recompilation, 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 as we need to preserve all function bodies and their callback wrapper implementations during the recompilation process. This approach also hinders interprocedural 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 given 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 unmark them as `external`. 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.

Handling Non-Atomic Loads/Stores

We initially adopted the fence insertion strategy formalized by Lasagne [98] for x86/x64 to handle memory access reordering at the IR-level. However, 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 provide different semantic guarantees than the acquire/release fences in LLVM IR. Except for sequentially consistent fences, fences in LLVM IR are required to be paired with another monotonically ordered atomic operations 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.

Ignoring the issue of non-atomic memory accesses, when recompiling x86/x64 programs without the aim of 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/x64 load to a sequentially consistent load instruction, and every x86/x64 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.

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. 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.

3.2.5 Non-Atomic Load/Store Optimization

Programs synchronize shared memory accesses through primitives such as barriers, locks, mutexes and semaphores. To achieve this, programmers often rely on external libraries for the implementations of such primitives, like those provided by `std::atomic`, pthreads, and OpenMP. But, they can also choose to implement custom or *implicit* synchronization primitives when available constructs are too slow or do not provide specific guarantees and control. Our key insight is that demonstrating the absence of implicit primitives in the binaries of data-race free programs can be leveraged to optimize the handling of non-atomic memory accesses during their recompilation. Note that as we recompile binaries for the same architecture, we care about memory access reorderings only at the IR-level.

Approach

Consider an analysis which identifies if given machine code ***does not*** implement any implicit synchronization primitives. We describe two scenarios where this is the case, and discuss the issue of shared memory access reordering during lifting in each case. Since all shared memory

accesses in a data-race free program must be synchronized,

- *the code exclusively uses synchronization mechanisms provided by external libraries*: In this case, the compiler is prohibited from reordering memory accesses across an external call as it is unaware of the possible side effects it may have. Therefore, it conservatively preserves the original access ordering irrespective of whether originally non-atomic loads and stores are marked as atomic..
- *there exist no shared memory accesses that need synchronization*: If no two threads are racing to access the same location (atleast one thread with a write operation, and another thread with a read/write operation), each memory access can be considered to be synchronized. In this case, IR-level reorderings maintain original program semantics.

Hence, if it can be shown that the given program does not implement any implicit synchronization primitives, we can relax the atomicity restrictions imposed on originally non-atomic loads and stores that prevent reorderings in the IR. To that end, we design a dynamic analysis based instrumentation pass to detect implicit synchronization primitives in machine code.

Prior work [50, 65, 78, 81, 105] identifies that the basic pattern necessary for implementing such a construct is a spinloop. Since multiple definitions of a spinloop exist in literature, we choose the most permissive one as described in AtoMig [22]. For each loop that we identify in the lifted IR, our analysis procedure checks if it is *NOT a spinloop*. We achieve this by showing that it is possible to exit the loop due to the influence of a local value that is, (1) not loop-constant and, (2) lacks external dependencies. A value is defined to have an external dependency if it depends on a shared memory access through some data flow.

Note that AtoMig detects spinloops to identify potentially racy memory accesses based on instruction influence analysis of the *spin controls*. They transform such operations to be sequentially consistent, to achieve the correct translation of programs written for a stronger memory model (TSO) to a weaker memory model (WMM). With Polynima, we aim to identify spinloops

```
@g = global i32 0
define void @samples() {
  %1 = alloca i32
  ; (a) Spinloop
  %op = load i32, i32* @g

  ; (b) Spinloop
  %5 = load i32, i32* @g
  store i32 %5, i32* %1
  ; ...
  %op = load i32, i32* %1

  ; (c) Spinloop
  store i32 1, i32* %1
  ; ...
  %op = load i32, i32* %1

  ; (d) Non-spinloop
  %7 = load i32, i32* %1
  %8 = add i32 %7, 1
  store i32 %8, i32* %1
  ; ...
  %op = load i32, i32* %1

  ; (e) Non-spinloop
  %3 = phi i32 [0, %entry],
           [%op, %back]
  ; ...
  %op = add i32 %3, 1

  }
```

Listing 3.5: Examples of spinloops and non-spinloops

to infer if the binary implements custom synchronization primitives as part of its code.

We illustrate the various cases through examples in Listing 3.5. We assume that the value `%op` is one of the operands for a loop termination condition and that the rest of the statements for each of the examples belong to a loop body.

Let us first consider the spinloop cases. (a) has a direct external dependency on `@g` and (b) has an indirect external dependency (through `store`) on `@g`. In both of these cases, external dependencies (`@g`) may be modified by other threads between loop iterations, which may invalidate assumptions about loop termination. This is usually how spinloops are implemented, where one thread spins until it can get exclusive access to a protected shared resource. (c), on the other hand, has a local store of a constant value `1`. Constant value stores do not affect loop termination across different iterations. For all of these cases, we mark the loops as potentially spinning.

Now, we consider the non-spinloop cases. (d) depends on a local store of a non-constant value. This is seen in cases where local program variables are accessed through memory loads and stores instead of through registers, such as for unoptimized IRs. Whereas, (e) depends on a loop-

modified value, which demonstrates a typical case of a locally-stored loop index being used to execute a fixed number of iterations. For both of these cases, we also ensure that there is no dataflow of an external dependency into `%op`. The influence of such an operand on the loop termination condition would be sufficient to mark it as non-spinning according to our definition above.

Analysis

We first recursively inline all lifted functions in the body of their callers to enable data flow tracking across procedure calls. Next, we perform the LLVM-provided loop simplification pass to restructure loops such that they have dedicated exit blocks. This enables the precise analysis of their termination conditions. Polynima's functional IR representation enables us to rely on LLVM's standard compiler passes to perform these transformations.

We annotate and instrument all memory access sites i.e. loads, stores, RMWs and CmpXCHGs to record the memory location and the access type i.e. local or shared. Polynima can differentiate between stack-local and shared memory accesses as we control the allocations for each thread's emulated stack. We then run the recompiled binary with a set of concrete inputs to record dynamic analysis information for the instrumented memory accesses. After merging data collected across various runs, we map each memory access site to a list of tuples, each containing the observed location and the access type.

Next, we iterate over all loops in the lifted IR and analyze them individually. Polynima performs an instruction influence analysis, which we model as a backwards dataflow analysis, for operands of each of the loop termination conditions. The goal here is to identify if any of the operand values are influenced by loop-modified local value. It is trivial to perform this analysis for values that are not influenced by memory accesses, by following their use-def chains in the IR. We typically observe this for source-level variables that are mapped to a register storage for the duration of

the loop body, such as loop indices. In this case, we benefit from lifting general-purpose registers as SSA values.

Performing this analysis for source variables that are stored in memory, and influence the loop termination condition, requires chasing memory loads and stores. We resolve these queries using the dynamically recorded information. For all sites that access *shared* memory locations, we assume an external dependency and discard checking further. For *local* accesses, we collect all intra-loop stores made to that location and trigger another backwards dataflow analysis for the stored values. If the stored value is, (1) not loop-constant and, (2) lacks external dependencies, we can assert that the loop is non-spinning.

Once we identify that ***all*** lifted loops in a binary are non-spinning, we conclude that the we can relax the atomicity restrictions for originally non-atomic loads and stores in the lifted IR. Figure 3.6 shows an overview of the entire process.

Limitations

False positives. Falsely asserting the absence of implicit synchronization may lead to unsound recompilation. Our approach can fail for programs that implement implicit synchronization primitives *without spinning*, i.e. using only loads and stores. This occurrence is uncommon, as construction of any non-trivial and wait-free synchronization mechanisms requires the usage of atomic read-modify-write accesses [54].

We do not support programs that use primitives based on sleep-based contention, and other asynchronous methods (e.g. signals and syscalls). But, we argue that through our dynamic analysis we can detect timeout-based synchronizing loops in atleast one running thread because, (1) we assume that the programs under analysis are data-race free, (2) for progress to be made toward program completion, atleast one of the spinning loops has to sucessfully exit. We did not find evidence of the use of any of the above mechanisms as part of our benchmarks.

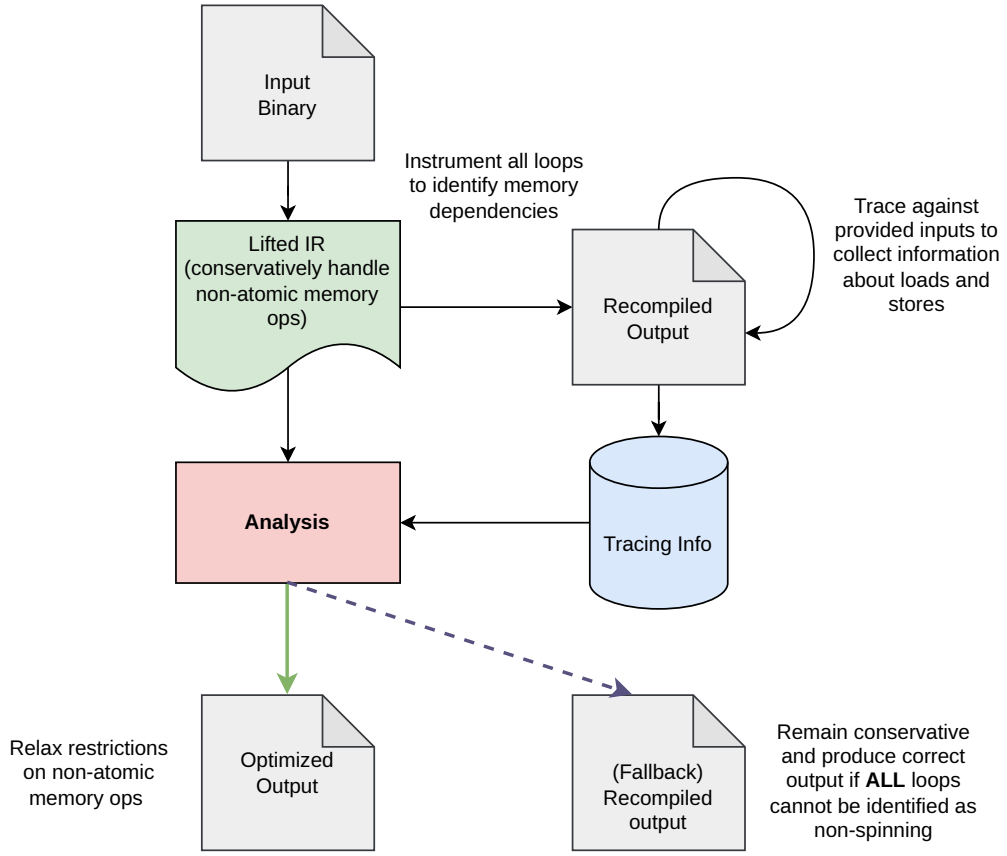


Figure 3.6: Overview of the dynamic analysis pass for optimizing the handling of non-atomic loads and stores.

False negatives. Our dynamic approach is limited by the loop-coverage that is achieved through the provided inputs. We may falsely identify that a program uses implicit synchronization if we, (1) do not realise a loop body during the execution runs, (2) identify a non-spinning loop to be spinning. Polynima also does not build precise happens-before relationships or perform lockset analysis to identify shared memory accesses that belong to a loop but are already synchronized. This may result in false negatives that are not resolvable through dynamic analysis. In such a scenario, we remain conservative and preserve restrictions on non-atomic loads and stores, possibly affecting performance but not correctness.

3.3 Evaluation

Our evaluation is guided by the following research questions:

RQ1: Does Polynima make available the transformation infrastructure, available as part of LLVM, to fix and improve 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: How effective is our load/store optimization approach?

RQ4: Does our hybrid control flow recovery approach improve state-of-the-art?

Environment and Software. We wrote a wrapper around the radare2 [13] 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 [76] tool, augments this representation with dynamically collected indirect control transfers. We then invoke the translator module, which is built on top of S^2E's [34] RevGen [33] utility. This provides us the infrastructure to translate individual machine code basic blocks to LLVM IR (LLVM 14). S^2E 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^2E. In the lifted IR, we stitch together lifted basic blocks to create functions based on the recovered control flows. Finally, the rest of our lifting pipeline builds on top of BinRec [12], 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, instruction recovery, lifting and (additive) recompilation of binaries. Users need only provide inputs that exercise control flows for the optional dynamic analyses. Writing patches

for binaries using Polynima is akin to writing a compiler-level pass for LLVM IR, with the option of adding a runtime component that can be linked in. A custom transformation pass can be integrated with the standard compiler infrastructure by registering it with LLVM's PassManager.

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, hyperthreading, 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`.

Comparison with other lifters. We tried running other state-of-the-art lifters identified in Liu et al. [74] to lift the binaries that we choose for our evaluation. The authors of RetDec [67] 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 [41] authors conveyed that the tool's main focus is binary lifting and its overall recompilation capabilities are experimental. To evaluate Rev.Ng [40], 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 [98], which builds on top of mctoll [115], 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.

To our knowledge, Polynima is the **only** binary recompiler that supports real world multithreaded programs while maintaining original program semantics.

3.3.1 Exploit Detection (RQ1)

We focus on detecting and mitigating CVE-2023-24042 [5], a recently discovered synchronization bug in LightFTP which enabled path traversal and possibly other security issues. The bug manifests because the variable (and the context) used to track the requested file name and the session user name is reused across the different threads creating a race condition. Below is the sequence of steps a malicious user would perform for a directory traversal exploit,

- Send the *LIST* command with an existing directory name as the parameter, writing the path in `context->FileName` and spawning a blocked handler thread.
- Send the *USER* command with a filename of choice (e.g., `/etc/passwd`) as the parameter, overwriting `context->FileName` with this value. Note that no checks are performed for this write.
- Connect to the data socket which unblocks the handler thread for the *LIST* command. The handler now uses the value stored in `context->FileName`, which has been overwritten in the previous step.

We identify that the program calls `stat` to check the file status in function `ftpLIST` before spawning the handler thread. The handler function `list_thread` then calls `opendir` to open a directory stream corresponding to the requested path and return a list of files. We write an LLVM pass which records and compares the path arguments passed to the `stat` and `opendir` calls. During benign execution of the program, both would correspond to the same value, but would be different in the case of an exploit.

The operator is enabled to take various actions in an exploit scenario with a Polynima-recompiled LightFTP binary. They may divert the code to a custom runtime handler, written in plain C/C++, similar to a "patch" in source programs. They could also choose to log the event for forensics or stop the server entirely. Since we lift external calls and their arguments, it is also possible to replace the value stored in `context->FileName` with the older value to protect against the

Table 3.1: Supported Benchmarks. Lasagne builds on top of mctoll.

Benchmark	Description	LOC	Polynima	Lasagne	McSema	BinRec	Rev.Ng
memcached [110]	Key-Value Store	24.4k	✓	✗	✗	✗	✗
mongoose [77]	Web Server	7.4k	✓	✗	✗	✗	✗
pigz [8]	Compression Tool	6.4k	✓	✗	✗	✗	✗
LightFTP [55]	FTP Server	2.4k	✓	✗	✗	✗	✗
Phoenix [97]	Data Processing	4.4k	7/7	5/7	0/7	0/7	0/7
gapbs [21]	Graph Processing	2.8k	8/8	0/8	0/8	0/8	0/8
CKit (spinloops) [11]	Sync. Primitives	1.3k	11/11	0/11	0/11	0/11	0/11

exploit. Also, the operator has complete control over the set of valid control transfers in the lifted IR. They may choose to completely disable certain allowed commands by either rewriting their handler implementations or by limiting the available targets of a jump-table style command dispatch.

The actual fix for this bug involved major changes, maintaining a per-handler context structure consisting of the file's username and path. We argue that in the cases where fixes for such bugs are unavailable due to lost source or lack of vendor support, Polynima's capabilities to generate a replacement binary are demonstrably valuable. The compiler pass and the runtime instrumentation code account for only about 70 lines of C++. Thus, we enable a *usable and powerful interface* for performing program-wide transformations that can leverage the LLVM compiler infrastructure.

3.3.2 Compatibility and Performance (RQ2)

We test Polynima on a large and functionally diverse set of binaries that comprises real-world utilities and benchmark suites listed in Table 3.1. *We report correct outputs across all the test cases that we run.*

memcached uses pthreads along with compiler builtins 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 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.

mongoose. We compile 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.

LightFTP. 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%.

Phoenix. Table 3.2 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. We use the provided small, medium, and large input datasets to evaluate performance of the recompiled binaries.

We first 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*. 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

Table 3.2: Performance of Polynima recompiled binaries on the Phoenix benchmark suite. Results in the NA column report performance after relaxing restrictions on non-atomic loads and stores.

Benchmark	O0	O0 FO	O3	O3 FO
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

were originally compiled with little to no optimizations for an older CPU version.

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

We evaluate all gapbs binaries (Table 3.3) 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 Phoenix i.e. close to original performances for unoptimized binaries and slowdowns for the optimized versions.

Performance Discussion. We use the geometric mean of the results for the unoptimized (`O0`) and optimized (`O3`) Phoenix and gapbs benchmark suites to compute the overall 1.07x slowdown. We now discuss the major reasons for degradation in recompiled output performance for optimized binaries (`O3`) in gapbs and Phoenix.

Recompiled output performs memory accesses which are part of the original binary onto an emulated stack, which helps Polynima remain general in its approach to lifting binaries. However,

Table 3.3: 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	0.66
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	1.55	0.83	1.17

note that most optimizations in the LLVM ecosystem are designed to work with an IR that contains program variables along with their type information. Since we do not recover this, 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 due to the non-optimal lifting of SIMD instructions and floating point operations. Polynima relies on QEMU [23] helpers to provide translations for such instructions, which are based on emulating them on the virtual CPU state. We only implement precise lifting of certain vector instructions, which introduces some performance slowdowns. For certain vector instructions, LLVM can resynthesize them into intrinsics after lifting, but this translation is not optimal.

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, 19 on average, 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. Polyn-

Table 3.4: 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

ima builds on top of BinRec which outperforms McSema and Rev.Ng recompiled binaries on single-threaded benchmarks [12]. As a result, we expect Polynima to perform better than or atleast as well as BinRec in comparison to them. 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 (AArch64), which we do not support yet.

ckit. ConcurrencyKit implements custom concurrency primitives using compiler builtins (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 number of clock cycles required) for each spinlock. Each individual test consists of 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. In Table 3.4 we report that the recompiled binary performance is close to the original in almost all cases, which validates our earlier claims of efficiency and correctness.

3.3.3 Implicit Synchronization Detection (RQ3)

Next, we evaluate the precision of Polynima's spinloop detection as well as the performance improvements that we derive due to the subsequent relaxation of non-atomic loads and stores. We first validate our approach on the various spinlock implementations in ConcurrencyKit as representative examples of *implicit* synchronization primitives. Then, we evaluate it on the Phoenix benchmark suite, which explicitly uses external synchronization primitives, where we benefit the most by proving the absence of implicit synchronization. Recompiled binaries are run against provided inputs with instrumentation that records information for all memory accesses.

False positives. We do not observe any false positives in the experiments that we perform.

False negatives. In *histogram*, we fail to cover one loop body that swaps data bytes depending on the endianness of the underlying architecture. Since no inputs we provide would cover this loop (on x86/x64), we manually analyze it as a non-spinloop and report the results.

We observe a false negative in the case of the *pca* binary, as it requires a precise happens-before analysis for proving that a certain loop is non-spinning. This does not affect correctness however, as we default to preserving atomic guarantees for all loads and stores. We still report the results after performing the atomicity relaxation to demonstrate the impact on recompiled binary performance.

True positives. Apart from the two cases mentioned above, we cover and check that all other loops from Phoenix are correctly identified as non-spinning.

True negatives. We correctly identify all spinloops in binaries compiled from the validation test suite for the various spinlock implementations in ConcurrencyKit.

We refer to Table 3.2 for performance discussion of this optimization. We observe that relaxing atomicity restrictions on non-atomic loads and stores leads to a notable improvement in

performance for nearly all test cases. The average speedup observed for unoptimized binaries is improved to 0.81x, further pushing the case for using Polynima as a post-release optimizer. Removing ordering restrictions enables off-the-shelf compiler optimizations to be more effective. Crucially, we observe astounding improvements in performance for the *kmeans* and *string_match* binaries after relaxing non-atomic loads and stores. Improvements are also observed for optimized binaries, where the slowdown is improved to 1.08x.

3.3.4 Lifting Time (RQ4)

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 support 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. Our prototype was unable to handle `403.gcc` and `483.xalancbmk` due to failed IR translation for certain superfluous code paths.

We report the total time taken to disassemble, trace, and recompile with Polynima in Table 3.5. We refer to the BinRec paper [12] 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

Table 3.5: Lifting Times (in s) the for SPEC INT 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	–

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.

On the other hand, for a program such as `445.gobmk` it is difficult for a static disassmbler to precisely resolve such a large number of indirect control transfers (1241). Recent work was unable to functionally verify McSema-recompiled binaries for more than half of the SPEC benchmark suite [74]. In this case, 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 3.7. 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

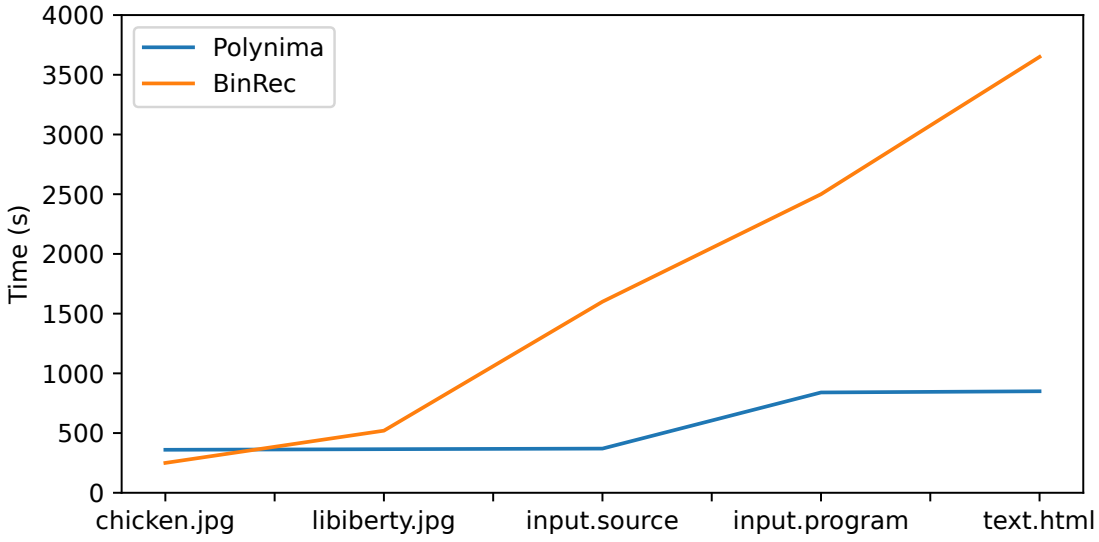


Figure 3.7: Lifting times for BinRec's Incremental lifting v/s Polynima's Additive lifting for `401.bzip2`.

to IR. Performing the IR translation *offline* is key for recompilation to scale to large 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.

3.4 Conclusion

With Polynima, we demonstrate a meaningful composition of static and dynamic techniques to perform binary recompilation of multithreaded binaries. We start with a sound static approach and then design optional dynamic analyses that incrementally refine the the lifted IR.

Chapter 4

StackBERT: Machine Learning Assisted Stack Frame Size Recovery on Stripped and Optimized Binaries

Chapter 3 introduced a practical and sound way to recompile multithreaded binaries. However, many of the off-the-shelf LLVM optimizations remain unavailable to modern recompilers because they fail to reason about the program stack.

This chapter presents StackBERT, a framework to statically reason about and reliably recover stack frame information of binary functions in stripped and highly optimized programs. The core idea behind our approach is to formulate binary analysis as a selfsupervised learning problem by automatically generating ground truth data from a large corpus of open-source programs. We train a state-of-the-art Transformer model with self-attention and finetune for stack frame size prediction. We show that our finetuned model yields highly accurate estimates of a binary function's stack size from its function body alone across different instruction-set architectures, compiler toolchains, and optimization levels. We successfully verify the static estimates against run-

time data through dynamic executions of standard benchmarks and additional studies, demonstrating that StackBERT's predictions generalize to 93.44of stripped and highly optimized test binaries not seen during training.

4.1 Motivation

Although stack symbolization of binary programs would enable many possible applications in binary rewriting, lifting, and recompilation, it currently remains as a largely open problem. There are several reasons for this: first, binary programs in principle are free not to use the system's call stack and arbitrary binaries that were produced using handwritten assembly, self-modifying code, or code virtualization obfuscation might explicitly opt not to. Second, even for binaries that are generated by standard compilers—and which consequently should adhere to the system's Application Binary Interface (ABI)—symbolizing stack accesses is uniquely challenging [27]. In fact, it can be nearly impossible for human developers to make sense of a program's stack management just by looking at stack traces post mortem: the Linux kernel developer community had to develop a dedicated stack metadata validator [95] and unwinder [61] to deal with mounting problems of garbled and unintelligible stack traces in bug reports and crash dumps.

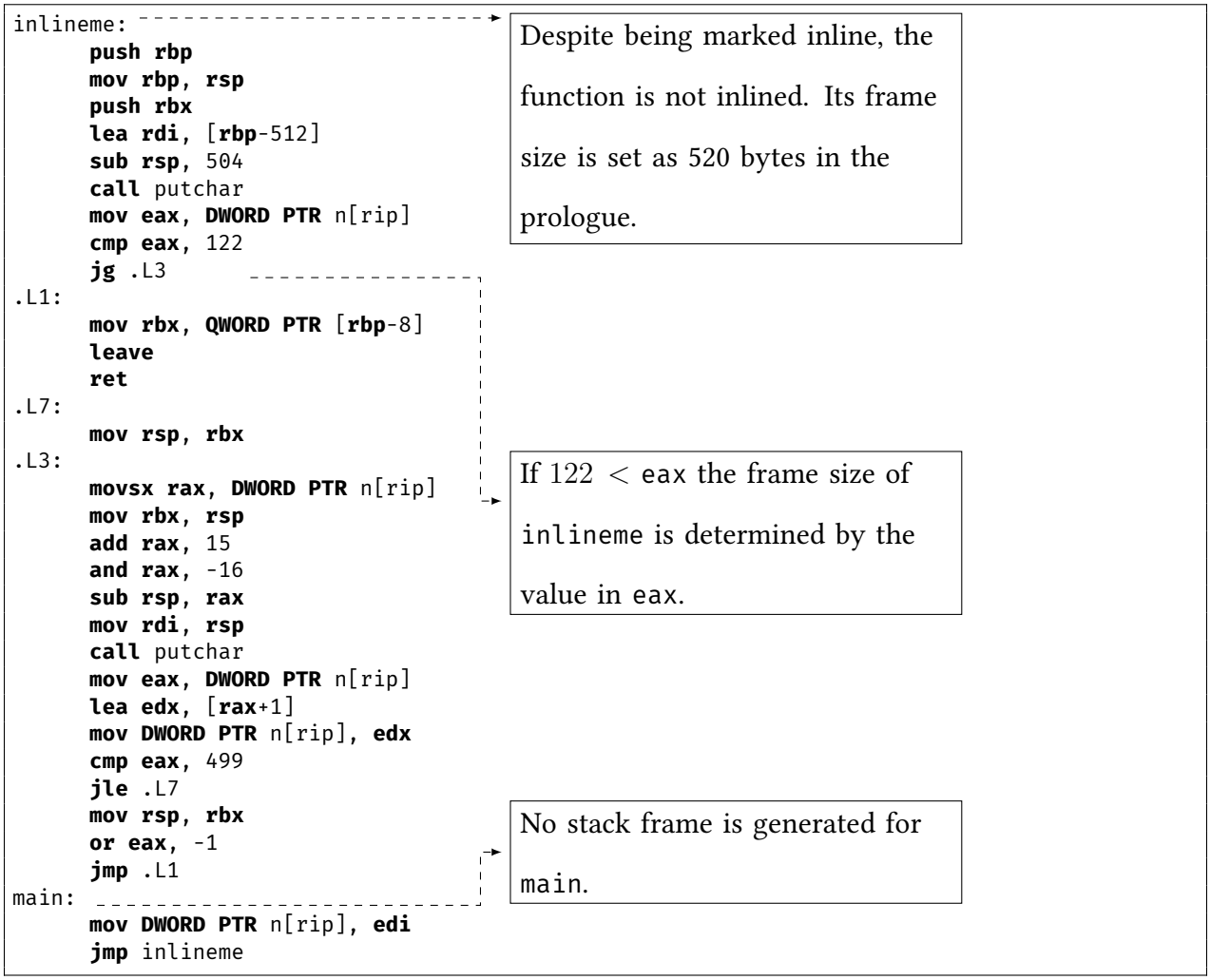
To illustrate some of these challenges concretely let us consider a simple source-code example in Listing 4.1. It contains two function definitions as well as a global integer variable definition. Since `main` takes parameters and also defines local variables, one might expect a dedicated function frame. Moreover, as the function called by `main` is annotated with the `__inline__` intrinsic, one might also expect its stack frame to be part of `main`'s stack frame, given this particular source code. However, if we look at the disassembly of the binary in Listing 4.2 that is generated by the two big compiler suites GCC (in version 11.1.0) and LLVM (in version 13.0.0) with only minor variations under modest optimization levels (i.e., -O2 and -ansi), we discover that neither of these assumptions hold.

```c
static volatile int n;
static __inline__ int inlineme(int * i) {
    int foo[123] = {};
    putchar(foo);
    if (*i < 123)
        return *i;
lt500:;
    char x[n];
    putchar(x);
    if (n++ < 500)
        goto lt500;
    return -1;
}
int main (int argc, char *argv[]) {
    int *b[10];
    n = argc;
    b[5] = (int *)&n;
    return inlineme(b[5]);
}
```

Listing 4.1: Example C program (compiles with `-O2 -ansi`).

The reason is that both compilers determine the stack layout of function `inlineme` to be incompatible with the stack layout of `main`, and hence, cannot inline the function. They do however determine `inlineme` to be tailcall optimizable and directly forward the local variable definitions through `main` via constant propagation. The result is that `inlineme` will be defined as a function symbol in the binary's symbol table despite being marked inline and never explicitly being called anywhere (only jumped to). Consequently, a crashdump of this program's execution if called with more than 121 command line arguments (e.g., with "`seq 122 | xargs ./example`") will not show the `main` function being called. The exact same issue also arises during debugging. If

this program required live patching in a production system, the call stack would not serve as a reliable source of its execution state. [1]



Listing 4.2: AMD64 assembly for the Example program generated by GCC (LLVM output is practically identical).

For this reason, one of the first steps in symbolizing stack accesses for binary code is to bound the size of each function's call frame. As we saw in the example above even for binaries generated by popular and widely used compiler frameworks with standard compilation options, many functions will not actually have a dedicated call frame, or its size may depend on the context. Functions that are inlined multiple times by the compiler may be inlined into the caller's stack frame with different layouts and sizes. For some functions the size of their stack frames may be

[1] This is why the Linux kernel requires `-fno-omit-frame-pointer` for producing reliable stack traces.

a range of possible values depending on the exercised control flow with an upper limit that can be computed at compile time. However, in general we saw that a function's stack frame may not be bounded at all and programs that call `alloca` (or one of its many variants) with a dynamically determined size argument or make use of Variable-Length Arrays (VLAs) can exhibit differently sized stack frames for different inputs. Since this type of runtime-dependent behavior can cause various issues with respect to compatibility, debugging, and memory safety it is often actively discouraged [68].

Assuming the stack frame size is at least bounded at compile time, there are several ways one could try to statically reconstruct an upper bound for frame sizes of function definitions for a binary program: (1) do the obvious thing and emit stack frame sizes during compilation (`"-fstack-usage"`), (2) count `push` and `pop` instructions, also considering all other modifications of the stack pointer (like `"sub rsp, 504"`) [73], (3) use the entries of the Canonical Frame Address (CFA) column in the function's `.eh_frame` table to try and determine the frame size, (4) collect all `DW_TAG_variable` and `DW_TAG_formal_parameter` entries, as well as registers written to the stack, calculate their respective sizes in bytes, and sum up the total (while also taking their stack offsets into account, which might overlap).

While the first option is probably the safest and most precise, since the compiler originally determines a function's frame layout, this requires source code and target compiler to both be available. It further requires recompilation to be an option for deployment—which may not always be the case even if source code and target compiler are available—and is therefore unfortunately the least generally applicable among all options listed above. In our example both LLVM and GCC correctly report the frame size of `inlineme` as 520 bytes but also mark it as "dynamic".

In contrast, the second option sounds pragmatic, but requires accurate code discovery and modeling of stack operations for each architecture individually—all of which come with their own set of complications. As for more complex binaries and instruction-set architectures a simple linear sweep may not actually be sufficient to discover all stack modifications and symbolically

	`-fstack-usage`	count	`eh_frame`	DWARF info
binary-only	✗	✓	✓	✓
accurate	✓	✗	✗	✗
general	✗	✓	✓	✗
complete	✓	✗	✗	✗

Table 4.1: Comparison of possible approaches to statically identify stack frame size of a binary function.

executing functions with global context and intricate control flows would require sophisticated input generation, this approach is constrained to relatively simple cases in practice.

The third option relies on metadata that is present in the vast majority of binaries (including stripped binaries), and hence, may seem like an attractive alternative. However, this requires CFA-entries that are relative to the stack pointer and the compiler may opt to represent CFA entries as base-pointer-relative (or even use general purpose registers), in which case they are useless for determining the stack size of the function statically.

The fourth option requires a debug build of the binary, as well as reliable and complete type information, which is usually not available.

We provide an overview in Table 4.1: in summary, all of the approaches we discussed are limited with respect to frame size recovery for binary programs and we discuss some of the practical implications of this in Section 4.4. For our approach, we aimed at combining the accuracy of a compiler-based solution (which requires source code) with the applicability of binary-only methods to determine frame sizes statically while forgoing the source code requirement. We implemented a simple baseline method for the remaining approaches outlined above as part of our framework to compare against the results using a learned program representation, which we present in the next section.

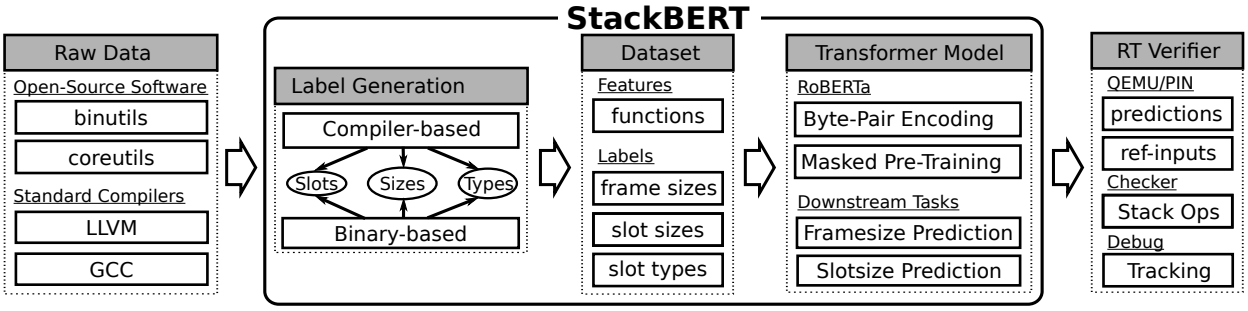


Figure 4.1: Overview of StackBERT: first, we automatically build and pre-process a large corpus of open-source software using two popular and widely used compiler frameworks. We then automatically extract both features and labels from the compiled artifacts. Next, we pre-train a state-of-the-art Transformer model (RoBERTa), using byte-wise masking of the function disassembly as pretext task. Since the collected label information are not usually present in stripped binaries, we then finetune the pre-trained model using one of our custom downstream tasks, by modeling a key stack symbolization problem as classification with binning. To accurately assess model accuracy we test and validate the model predictions against unseen inputs and also verify its outputs dynamically using standard benchmarks with reference inputs.

4.2 Design and Implementation

In this section we present the design and implementation of StackBERT. Our main goal is to statically recover stack frame sizes of binary functions. For this, we first train a Machine Learning model based on the popular Transformer architecture using over 600,000 compiled function bodies with masked byte prediction. We then finetune the model to recover an upper bound on each function's frame size.

4.2.1 Overview

We designed StackBERT to be able to operate in the form of a continuous, supervised learning pipeline that consists of a number of components and present an overview in Figure 4.1: ① a collection of widely used open-source programs (binutils, coreutils) built with standard compilers (GCC and LLVM) using a number of different optimization levels, ② a label generator, including compiler-based label generation plus tools for parsing ELF binaries using debug information as a

baseline implementation for function frame size recovery, ③ a training set of pre-processed and automatically labeled data, ④ a state-of-the-art Transformer model architecture, pre-training task setup, as well as custom finetuning task through binning, ⑤ a dynamic verifier for standard benchmark programs using reference inputs.

4.2.2 Technical Challenges and Baseline Analysis

As outlined in Section 4.1, there are several ways in which one might try to obtain a frame size per function statically from a binary in principle. To get a sense of how well such "conventional" approaches perform (and since none of them seemed to be publicly available for general purpose architectures) we prototypically implemented them as a baseline for our approach (cf., "Binary-based Label Generation" in Figure 4.1). We make several simplifying assumptions such as dealing with benign, compiler-generated executables that are unstripped, formatted as standard ELF files, that may or may not be built using debug information.

First, we utilize the popular `pyelftools` Python library [25] to parse the binaries and read the contents of all executable sections, as well as symbol information (`.symtab`), call frame information (`.eh_frame`),[2] and (if present) debug information (`.debug_info`. beta beta still).

Next, we associate each call frame information with their respective symbol definition. The frame descriptor table consists of entries that are intended to be used for scenarios such as frame unwinding, debugging, and core dumping. An individual entry specifies how the state of the program can be interpreted at any given point during that procedure's execution. For instance, if a register was saved to the stack prior to calling another function the caller's frame table should contain an entry that specifies at which location within the function's frame that value was stored. The syntax for parsing entries is quite complex, due to a finite-state machine encoding for saving

[2]We note that although an `.eh_frame` section is not required by ELF, it is not removed by the `strip` tool, and hence, typically present even in stripped binaries. For binaries that do not have an `.eh_frame` section, it can be synthesized automatically [19] from a function's disassembly.

space. It is specified as part of the DWARF Debugging Information Format Standard. Although the call frame information in the `.eh_frame` does not represent debugging information, it largely follows the same format.

We parse register rule expressions and generate a preliminary frame layout based on rules that specify a dedicated stack location for any object within the function's frame table. Finally, we collect and propagate any type information that is contained (or can be assumed, e.g., based on the ELF's target architecture) in the binary, to obtain a number of bytes for all stack objects. Although stack slots semantics may depend on the program counter and can, e.g., be reused to hold multiple objects of varying types and sizes at different times, we ignore complex control flows for calculating the maximum frame size. We emit three different estimates for each function's frame size, according to (i) the function's disassembly, (ii) the function's canonical frame address, (iii) the maximally possible sum of all objects with a dedicated stack frame location and well-defined size in bytes. We present the numbers obtained using these three different baseline approaches and a comparison against the estimates obtained using our finetuned Transformer network in Section 4.3.

4.2.3 Pretraining: Masked Byte Prediction

The core idea and main component within StackBERT is our pretraining and finetuning of a state-of-the-art Transformer model to learn instruction set semantics in a self supervised manner, while supporting a variety of different instruction set architectures—like AArch64 and AMD64—as well as potential downstream tasks. In this way, StackBERT is able to readily support prediction tasks largely independently of build toolchain and other metadata typically used in conventional binary analysis approaches. Our generic pretraining setup has two important consequences: first, we are able to leverage the large body of existing software for training without expensive collection of labeled data. Second, the resulting model only requires a minimal amount of information during

inference, i.e., binary code of a function.

We pretrain the model using a self-supervised token prediction task, where each token is equivalent to a byte of instruction opcodes as present in the compiled binary function body. Given a sequence of 512 bytes per sample we randomly mask a byte in the sample with a probability of 20%. The model then has to predict the masked bytes in the sample, minimizing over a variation of the cross-entropy loss as objective function, tuned for masked language prediction (see Eq. 1). We refer to Section 4.4 for additional discussion on our pretraining setup. For pretraining, we use the Adam optimizer with a learning rate of 10^{-4} and polynomial decay scheduling.

$$L = -\sum_{i} y_i log(\hat{y}) + (1 - y_i) log(1 - \hat{y}_i) \tag{4.1}$$

4.2.4 Downstream Task: Frame Size Prediction

For our downstream task, we use the function body as input sequence to our model and the maximum size of its stack frame as output. Bounding the size of a function's stack frame represents an important part of the stack symbolization problem. Potential applications of frame-size recovery include binary instrumentation and rewriting, run-time patching, as well as recompilation after binary lifting to enable use of the native stack. We model frame-size prediction as a classification problem via *binning*: in particular, we define nine different classes that correspond to stack frame sizes in the range of 8 to 2048 (see Section 4.4 for additional discussion). We use the Adam optimizer with an initial learning rate of 10^{-5} and polynomial decay scheduling to finetune the pretrained model using the *sentence prediction* task, calculating the maximal score of the function body over all classes. We utilize groundtruth labels collected from the two big compiler toolchains GCC and LLVM in recent versions to calculate the loss. We detail our label collection process for this finetuning step in the next section.

Task	Samples	
Dataset	Binaries	Frame Sizes (Mean/Std)
Training		
allutils-GCC-AMD64-O3	124	36,646 (95.24 / 702.64)
allutils-GCC-AMD64-O2	124	38,115 (85.27 / 684.60)
allutils-GCC-AMD64-O1	124	40,105 (82.17 / 668.48)
allutils-GCC-AMD64-O0	124	52,594 (94.40 / 589.58)
allutils-LLVM-AMD64-O3	124	27,660 (90.16 / 853.00)
allutils-LLVM-AMD64-O2	124	27,783 (89.54 / 851.02)
allutils-LLVM-AMD64-O1	124	27,827 (90.08 / 850.12)
allutils-LLVM-AMD64-O0	124	38,066 (123.81 / 770.62)
allutils-GCC-AArch64-O3	124	41,035 (97.38 / 600.54)
allutils-GCC-AArch64-O2	124	42,865 (87.01 / 582.98)
allutils-GCC-AArch64-O1	124	45,650 (84.50 / 565.93)
allutils-GCC-AArch64-O0	124	60,502 (92.00 / 488.92)
allutils-LLVM-AArch64-O3	124	43,200 (90.49 / 585.35)
allutils-LLVM-AArch64-O2	124	43,512 (89.87 / 583.13)
allutils-LLVM-AArch64-O1	124	43,580 (91.49 / 582.37)
allutils-LLVM-AArch64-O0	124	62,450 (118.97 / 605.01)
Testing		
SPEC2017-GCC-AMD64-O0	23	27,885 (132.95 / 926.94)
SPEC2017-LLVM-AMD64-O0	23	73,627 (92.35 / 645.73)
SPEC2017-GCC-AArch64-O0	23	28,981 (153.31 / 1049.25)
SPEC2017-LLVM-AArch64-O0	23	70,951 (101.42 / 653.41)

Table 4.2: Overview of our Training and Test Datasets (O1-O3 omitted for brevity on the test set)

4.3 Evaluation and Results

We conduct all of our experiments using Google's Colab cloud compute engines with GPU support, the baseline only uses CPU instances.

4.3.1 Dataset

To train our model on a representative number of input samples we automatically generate a dataset of compiled binaries and corresponding labels consisting of real-world programs. In particular, we use the popular and widely used `GNU binutils` as well as `GNU coreutils` set of system programs, combining them into our `allutils` corpus of compiled programs (we present an overview in Table 4.2). We compile both program collections with two mainstream compilers, GCC 11.1.0 and LLVM 13.0.0 and collect groundtruth labels during the compilation process by adding the `-fstack-usage` flag. Compiling with two different compiler frameworks targeting two entirely different ISA's, using a number of varying optimization levels results in a natural diversification of the dataset. However, creating even bigger training sets is entirely possible using our automated data pipeline using these two compilers, possibly including other target architectures. While the outputs of `-fstack-usage` may be unreliable in the presence of link-time optimizations (LTO) [86], SPEC2017 does not compile with LTO by default and does not use it for reference inputs. For this reason, our dataset does not contain any link-time optimized binaries. Each compiler compiles the program for two different architectures AMD64 and AArch64 and across 4 different optimization levels `O0 - O3`. We design experiments to stress that our approach is compiler and architecture agnostic.

The range of frame sizes in our collected dataset is quite high, with a maximum frame size for optimized binaries of 65640 bytes when compiling Coreutils with LLVM (and 65616 bytes when compiled with GCC respectively), which represents more than 16 full pages of memory. The function allocating this large stack frame is `cksum_pclmul` in `src/cksum_pclmul.c`. [3] Since the distribution of the data is long-tailed (cf., stackbert/figures 4.2 and 4.3) with very few examples for sizes larger than a couple hundred bytes, we do not use functions with frame sizes over 8192 bytes for training.

[3] `https://github.com/coreutils/coreutils/blob/4edad9e1210dfaa4c8630bad16d0b2e6090de790/src/cksum_pclmul.c`

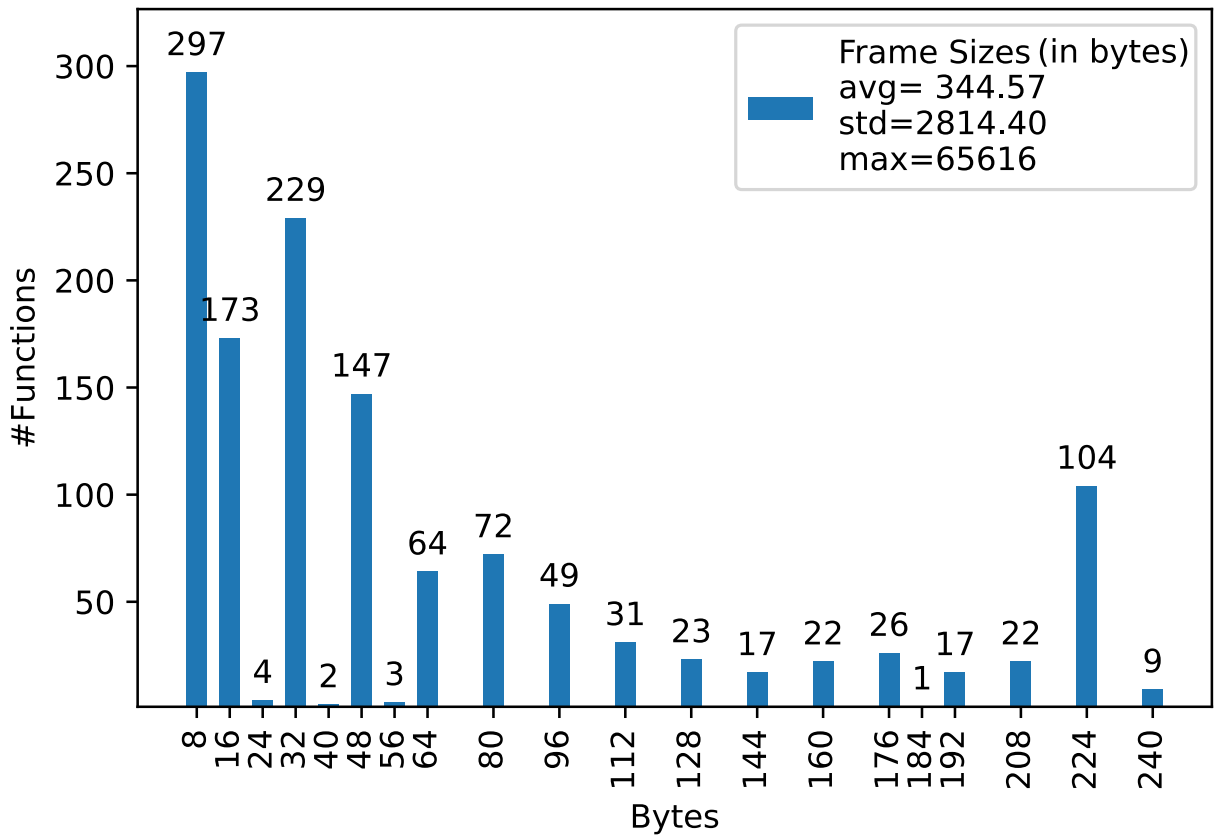


Figure 4.2: Coreutils Static Frame Sizes (-O3 GCC 11.1.0; x-axis truncated for brevity, since distribution is long-tailed)

We use the SPEC 2017 benchmark suite as a "holdout" dataset for testing our approach. We exclude binaries which involve FORTRAN based code from the evaluation.

4.3.2 Training Details

As described in Section 4.2, we use the RoBERTa base model architecture for our experiments as provided in the Fairseq [87] PyTorch library developed by Facebook, which has around 125M trainable parameters. As is common practice for language models with self-attention mechanisms, we first use a pre-training task to automatically learn a useful representation of the raw binary data and subsequently finetune the model on a downstream task.

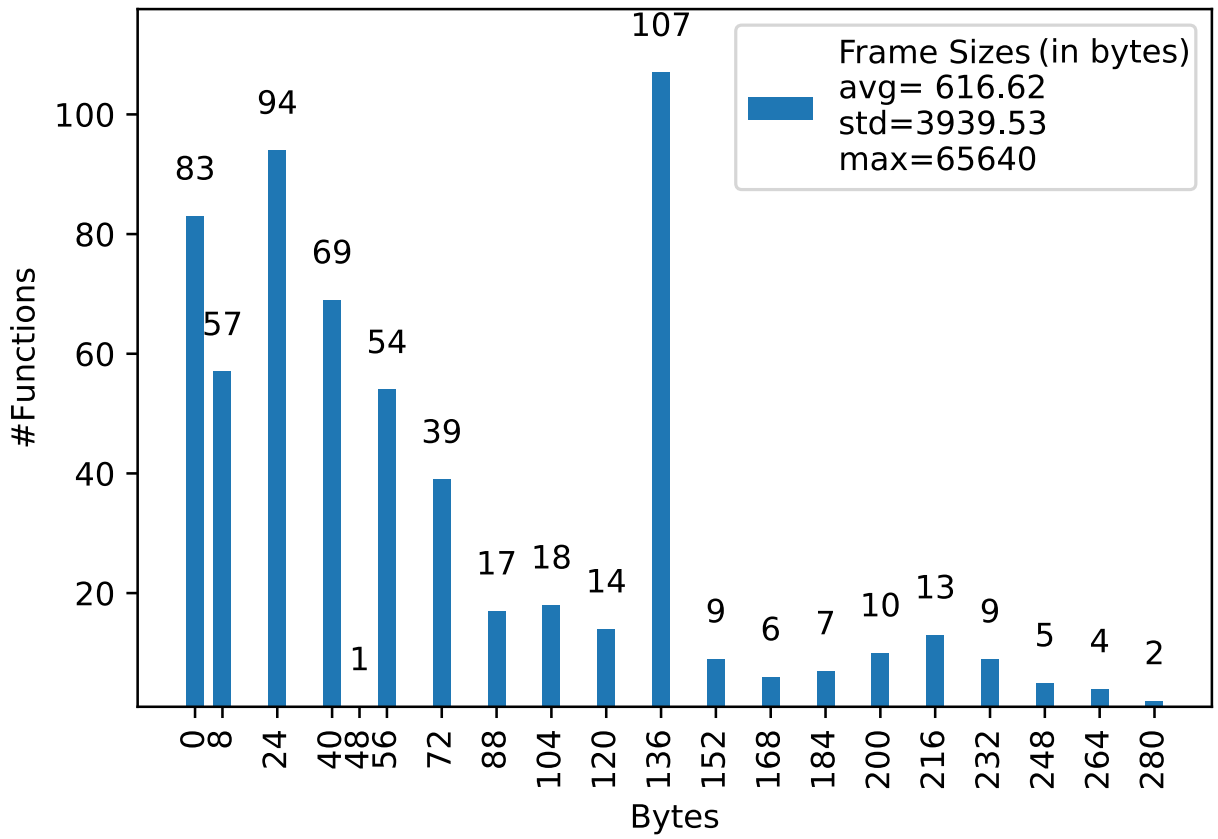


Figure 4.3: Coreutils Static Frame Sizes (-O3 LLVM 13.0.0; x-axis truncated for brevity, since distribution is long-tailed)

For the downstream task we use binning with frame sizes spanning multiples of two (starting from 8, 16, 32, 64, etc.) and provide a negative log likelihood loss as defined by the pre-defined sentence prediction task in Fairseq. For each of the models that we train, we use distinct datapoints for the pretraining and finetuning tasks. This ensures that the pretraining task does not get access to any of the labeled data used for the finetuning task.

We pretrain the models for 30 epochs and finetune them for 15 epochs. We observe that the finetuning task converges after 4-5 epochs of training.

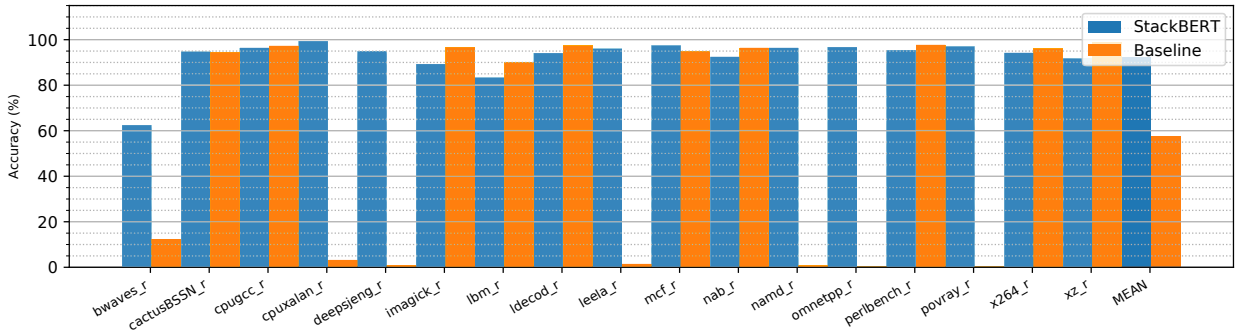


Figure 4.4: Accuracy of StackBERT vs. Baseline frame size predictions for SPEC2017 (on `-O3` AMD64)

4.3.3 Baseline Results

As explained in Section 4.2.2 we leverage a combination of `pyelftools` and `dwarf_import` Python libraries to parse the binaries and obtain static estimates of per-function frame sizes using our baseline analysis tool. It is also important to reiterate that our baseline requires unstripped binaries, and will not be able to generate predictions if certain metadata (such as the symbol table and the frame table) is not available. Even with all relevant metadata available the baseline analysis fails to make predictions in a number of cases for our training and test sets due to parsing errors such as unsupported DWARFv5 tags, reaching a total accuracy of 53.86% on SPEC2017 compiled for AMD64. We also evaluate the baseline analysis on SPEC2017 compiled for AArch64, where we observe a substantial drop to less than 20% mean accuracy (cf., Figure 4.6). The main reason for the substantial drop in performance appears to be the increased usage of DWARF constructs that are not supported in libraries utilized by our baseline implementation, as well as inferior general support for AArch64 binaries. Due to the complexity of the analysis (section parsing, type propagation, frame-table assignment) as well as the overall size of the binaries generating predictions with our baseline for the entire testset takes roughly 45 minutes and consumes up to 12GB of RAM.

4.3.4 StackBERT Results

Next, we evaluate the finetuned model on SPEC2017 binaries—which were never seen during training. In contrast to the baseline analysis, which relies on metadata embedded in the binary to predict frame sizes, StackBERT is able to generate predictions from the raw binary disassembly of a function body alone. This means it can make predictions even in absence of additional information and only requires raw byte inputs of the binary code, making predictions in a number of cases our baseline cannot cover. We plot the results of its predictions in blue in stackbert/figures 4.4 and 4.6. On average, StackBERT achieves an accuracy of 93.44% on SPEC2017 compiled for AMD64. Even for highly optimized binaries frame size prediction accuracies never fall below the 50% mark, and drop below 60% for just a single case. On AArch64, StackBERT's mean accuracy remains high at 93.46%.

We would like to highlight that the finetuned model we present here was trained on both architectures, i.e., AMD64 and AArch64. However, during our experiments we also trained a number of dedicated, architecture-specific models to make predictions for each architecture separately. Results for the same are detailed in Table 4.3.

Interestingly, the jointly trained model achieves an average accuracy that is around 3% higher than its architecture-specific counterpart for AMD64. We conclude that StackBERT manages to learn instruction-set semantics across different architectures automatically through our self-

Table 4.3: Mean accuracies across the SPEC2017 benchmark suite for the different models across optimization levels

Model Type / **Evaluation Dataset**	**AMD64** / **AMD64**	**AArch64** / **AArch64**	**Unified** / **AMD64**	**Unified** / **AArch64**
O0	90.35%	96.58%	93.56%	96.11%
O1	92.72%	95.73%	95.32%	92.85%
O2	90.87%	94.85%	92.38%	92.50%
O3	91.34%	93.75%	92.49%	92.38%
mean	91.32%	95.23%	93.44%	93.46%

supervised byte prediction pretraining.

While overall inference time depends on the size and complexity of the binary, StackBERT usually manages to predict stack frame sizes for individual functions in a matter of seconds.

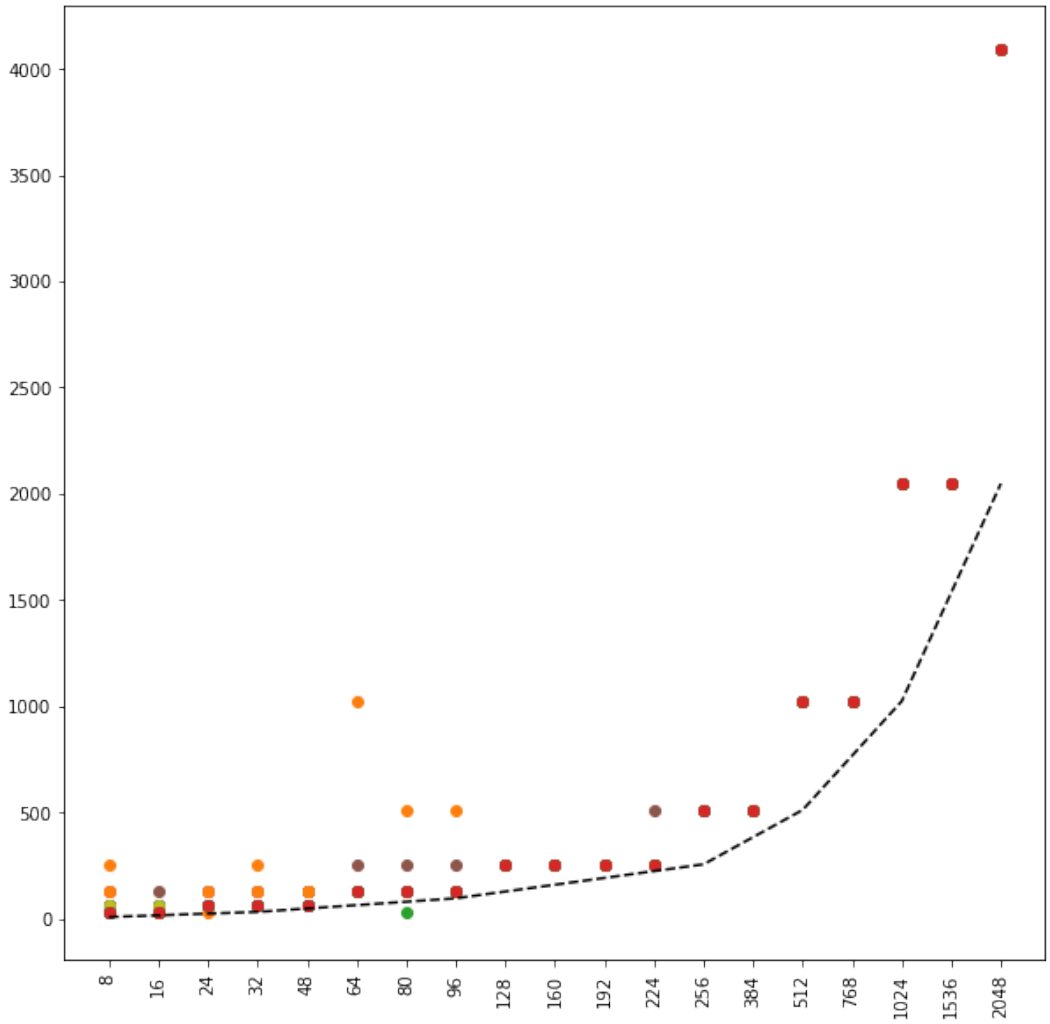


Figure 4.5: Predictions made by StackBERT under targeted modification of frame sizes continue to remain correct.

4.3.5 Additional Experiments

We conducted additional experiments to evaluate the fidelity of the learned representation by deliberately modifying stack sizes of binary function bodies. We chose the `mcf` binary from SPEC

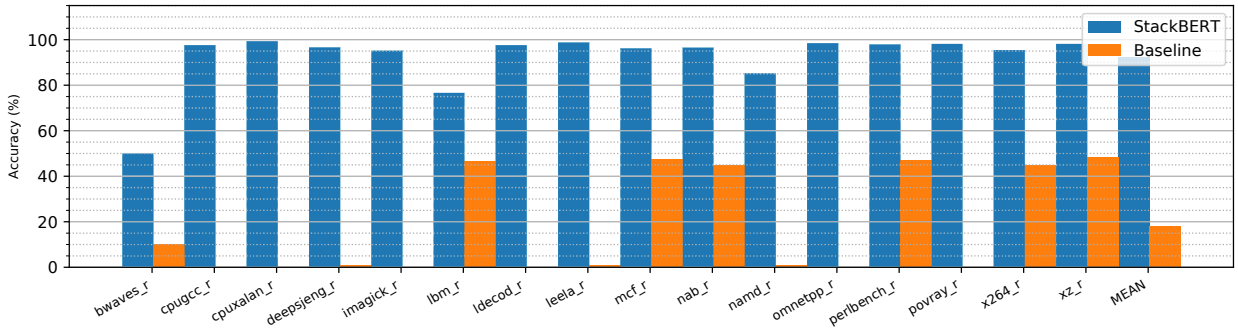


Figure 4.6: Accuracy of StackBERT vs. Baseline frame size predictions for SPEC2017 (on `-O3` AArch64)

2017 compiled with `O0` optimization using GCC 11.1.0 for AMD64 for this experiment. From this binary, we randomly picked function bodies containing direct manipulation of the stack pointer (e.g., `sub rsp, 0x40`). We modified the operand of these instructions by subsequently drawing integer values from a fixed range of numbers between 8 and 2048. We then predict the function's stack frame size after each individual modification. The result is depicted in Figure 4.5. The dashed black line shows the range of values that was drawn from the list. The dots show the predictions by StackBERT; as depicted only 0.22% of the predictions lay below the dashed line, meaning that a vast majority of the predicted values remain correct. We conclude that the trained model accurately learns which instructions will affect frame size through our pretraining and finetuning tasks. We also infer that the trained model learns quantitative relationships of tokens within the instruction and their relevance for the frame size of the containing function.

Last but not least we verified all of the correctly labeled predictions made by StackBERT for the SPEC2017 binaries by executing the respective binary and recording the stack frame size for each executing function using the PIN tool. We run the binaries with the test input provided in the benchmark suite to validate frame sizes of functions covered as part of the trace. While this severely impacted the execution time of the program, we were able to verify correct predictions of each of the function's frame size by checking that none of the static estimates exceeded the recorded runtime values.

4.4 Discussion and Future Work

In this Section we briefly discuss assumptions, pretraining, downstream task and binning, as well as possible applications.

As mentioned in the beginning, static binary analyses typically range from complex to generally unsolvable [56, 107]. For this reason, we constrain the stack symbolization tasks mentioned in this paper by making several basic assumptions: we assume standard ELF files that can be loaded by general purpose operating systems (i.e., the ELF magic is set and the file format conforms to the existing standards). We further assume that the binary files contain several standard information, such as a symbol table and correct function boundary information. While this may not always be the case in practice (for instance, because the binary was stripped) function boundary identification and symbol recovery are complimentary to StackBERT and several existing approaches demonstrate that such information can be recovered from stripped binaries in practice [10, 18, 93, 100]. We refer to Section 4.5.1 for an overview of existing methods. For our baseline analysis we further assume that the machine architecture matches the target architecture specified in the ELF file and that section headers and parameter settings are provided and accurate. While StackBERT does not rely on any metadata in the binary, our baseline analysis checks for a number of binary features in ELF files containing debug information, such as inlined calls, nested inlining, type information, variable and parameter declarations, and the baseline analysis assumes all of these to be correct.

For pretraining, it is noteworthy to mention that we also experimented with an instruction-based pretraining method, where we applied a similar objective as mentioned in Section 4.2.3 to disassembled byte sequences by masking all bytes belonging to an individual instruction at the same time. However, this instruction-wise pretraining method actually performed worse in our initial tests compared to the byte-wise masked pretraining in the subsequent downstream task. For this reason, we used the masked language prediction task as desdribed above. Similar to related

work [93] we hypothesize that pretraining lets the model learn an instruction-set architecture's (ISA) semantics prior to identifying the task-relevant bits, such as the instructions and operands that relate to the stack pointer. We note that truncating inputs to 512 bytes is a consequence of the underlying model architecture. In our case, this is not problematic as compilers generate instructions that define a function's frame size in the function prologue, i.e., at the beginning of the function body typically within the first few dozen bytes. However, should a function prologue exceed this limit in the future we could adopt model variations [24, 63, 118] that forgo input truncation.

Our downstream task uses binning to model stack frame size recovery as a classification problem. Our rationale for binning stack frame size predictions is two-fold: first, due to data alignment and performance optimizations compilers already tend to favor certain stack frame sizes over others. Second, language Transformer models with self-attention (like RoBERTa) are known to exhibit limitations with regards to counting and regression [26, 52]. However, there is no fundamental reason for binning frame sizes and we believe that with additional implementation and experimentation a regression model for frame size prediction (possibly using a modified architecture) should be feasible in principle.

Ideally, we would like to extend our existing downstream task to predict additional information about a function's stack frame, such as the number of individual objects and their respective sizes. While we anticipate StackBERT's design to be able to handle these additional tasks without requiring any modifications, the main challenge for this extension lies in the ability to gather highly accurate ground truth data. clang implements a machine function pass (accessed with flags `-Rpass-analysis=stack-frame-layout`) that prints out a textual representation of stack slots and outputs any debug information thats maps source variables to those slots. We anticipate that this information can be used to build more precise models.

We envision our results to be broadly useful for recompilation, since knowing a function's frame size while lifting it would enable breaking the global stack array into function-local arrays. Knowing an upper bound on the function's frame size should suffice to enable this use case: binary

recompilers can use this information to lower emulated, function-local stack frame arrays during recompilation to utilize the native stack, potentially yielding huge performance gains. According to our evaluation results in Section 4.3 more than 90% of binary functions in SPEC2017 could potentially have been rewritten to make use of the native stack in the recovered binary, rather than emulating stack accesses.

4.5 Related Work

Automated static analysis of binary programs has been studied for nearly three decades and a large body of literature on the topic exists. In the following, we provide a brief overview and comparison against relevant approaches.

4.5.1 Machine Learning for Machine Code

Machine learning (and in particular "Deep Learning" [70]) demonstrated significant progress over the past 10 years in a number of challenging and complex domains such as natural-language processing, computer vision, and robotics that have traditionally proven difficult for conventional software. It is thus perhaps unsurprising to see increasing adoption of ML-based approaches in other domains and a number of recent approaches propose to apply ML to static program analysis tasks. Since this area is rapidly growing we focus on a direct comparison with approaches that explicitly target *binary analysis* in this section.

Binary Function Identification

Several works tackled binary function identification using ML:

Byteweight [18] was one of the earliest examples that aimed at a data-driven approach to binary analysis. They propose a learned prefix tree representation for disassembly with normalized immediate values to classify function start addresses, tackling the function identification problem and beating IDA Pro (the state-of-the-art at the time) by an order of magnitude, establishing a new baseline. Shin et al. [100] were the first to demonstate neural networks for function identification, showing overall improvements in training and inference time over Byteweight (which was not based on neural networks) while keeping similar performance using a recurrent architecture with byte-wise one-hot encoding. Function identification approaches are complimentary to StackBERT, since we assume function boundaries to be known (e.g., through `.symtab`). Gemini [114] proposes to train a control-flow-based embedding for the purpose of binary function similarity detection, demonstrating order of magnitudes speedup in training and inference, as well as improved accuracy. Function similarity detection is orthogonal to our approach, but generally useful for malware classification and debloating. FUNCRE [10] tackles function inlinining detection by following up on an earlier approach [93]. Detecting if a function was inlined at a particular code location represents an important sub-task in analyzing highly optimized binaries. They are able to improve the F-score of state-of-the-art approaches in function inlining detection by about 3%.

Since we assume function symbols to be known for our frame size and layout prediction tasks, function identification approaches are complimentary to StackBERT.

Decompilation

An increasing number of ML-based binary analysis approaches aims to recover source-level information (commonly called "decompilation"), such as variable and function names, function signatures and line numbers, as well as high-level code constructs like loops, conditions, and switch statements:

Debin [53] uses a lifted binary intermediate representation to predict source-level debug information. They propose a conditional-random-field-based graphical model using factor graph representations and bayesian inference for structured prediction of likely source-level variable types and names given a particular binary program. Coda [46] presents a neural network architecture to decode a binary into an Abstract-Syntax Tree (AST) of a high-level source language, that is then iteratively refined using the target binary in a second step, achieving 82% program recovery accuracy on short, custom benchmarks built without optimization (-O0). N-Bref [47] proposes a dedicated structural transformer architecture using an assembly encoder, an AST encoder, and decoder. They demonstrate improvements of 6.1% and 8.8% accuracy in datatype recovery and source code generation respectively using short program snippets. Punstrip [91] combines probabilistic fingerprinting with a graphical model to learn relationship between function names and binary features. They are able to predict semantically similar function names based on code structure for standard library functions. Perhaps closest to our approach is XDA [93], which proposes a transfer-learning-based disassembly framework that uses masked language modeling as a self-supervised pre-text task to decompilation. The authors evaluate their approach on function boundary identification and code discovery as downstream tasks, using standard benchmarks on x86 and AMD64, achieving 99.0% and 99.7% F1 scores respectively.

Decompilation is orthogonal in principle to StackBERT, since our main goal is to recover low-level stack memory layout of a given binary function—which none of the existing approaches target, support, or evaluate.

Binary Type Inference

Another line of work is learned type inference for binaries. For example, Eklavya [35] proposes use of neural networks for function signature recovery by solving two tasks: function argument count and argument type recovery. They achieve an accuracy of 84% and 81% for both tasks respectively using a recurrent architecture with an instruction-wise, skip-gram-based word em-

bedding model that is trained separately. Function signature recovery partially overlaps with function frame recovery, as parameters may be passed via the stack. However, in practice many optimized functions do not receive parameters via stack accesses but through registers instead to increase runtime performance. Inferring types for local variables—which occupy the majority of function frames in our dataset—is not supported by Eklavya.

Stateformer [92] improves upon these results by introducing a custom pretraining task to statically approximate instruction execution effects using Neural Arithmetic Units. They then fine-tune their model to infer variable type information, adding several functional aspects (floating point, signedness, pointer types) as well as boosting overall accuracy over Eklavya [35] by 13% and achieving an average F1 score of 77.9% across architecures and optimizations.

Learned Models for Binary Execution

Finally, Ithemal [82] proposes to model execution timing aspects of complex instruction set architectures statically using an LSTM-based neural network architecture. They demonstrate that cycle-accurate throughput predictions can be learned efficiently for modern microarchitectures, significantly improving over the prior state-of-the-art (which does not use machine learning). Modeling timing aspects of instruction sequences represents an interesting but orthogonal task in static binary analysis.

4.6 Conclusion

With StackBERT, we show that reasoning about the maximum frame size of stack frames in binary programs is hard and provide a learned, static heuristic to achieve the same.

Chapter 5

What You Trace is What You Get: Dynamic Stack-Layout Recovery for Binary Recompilation

Chapter 4 introduces a static, learning based heuristic to recover the maximum bound on the individual function stack frame sizes in binaries. However, the recompiled binary may exhibit faults at runtime if a frame sizes are underapproximated by the model. For instance, stack local accesses may reach into the caller frame and change original program semantics.

In this chapter, we present a novel approach, What You Trace Is What You Get (WYTIWYG), to recover function-local variables within lifted 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. Improving upon StackBERT's heuristics-based technique to predict the perf-function stack frame size, WYTIWYG's dynamic analysis is sound for the set of inputs that is is traced against. Our approach is fully automated and preserves functionality for user-provided inputs. We demonstratate that WYTI-

WYG makes it possible to leverage the full potential of the compiler ecosystem to reoptimize legacy binaries through an extensive set of careful evaluations on the SPECint 2006 benchmark suite.

5.1 Motivation

5.1.1 Drawbacks of Stack Emulation

Lifting the stack as an opaque byte array severely limits the recompiler's ability to reason about the program. Consider the assignment and use of `ptr` in lines 4 and 7 of Figure 5.1. Since out-of-bound accesses are undefined behaviour, the source-compiler can assume that the access to `b` in line 5 cannot alias the value of `ptr`. This information is lost during compilation. Now, in order to infer that the loaded value of `ptr` in line 7 is identical to the spilled value in line 5, the recompiler has to prove that `ebp-44+f3(24)*8` cannot alias with `ebp-12`. Depending on the complexity of the operations involved in the address computation, this analysis quickly becomes challenging.

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 the primary cause limiting the efficacy of program analyses and transformations. Liu et al. have confirmed this finding in their study of binary recompilers [75].

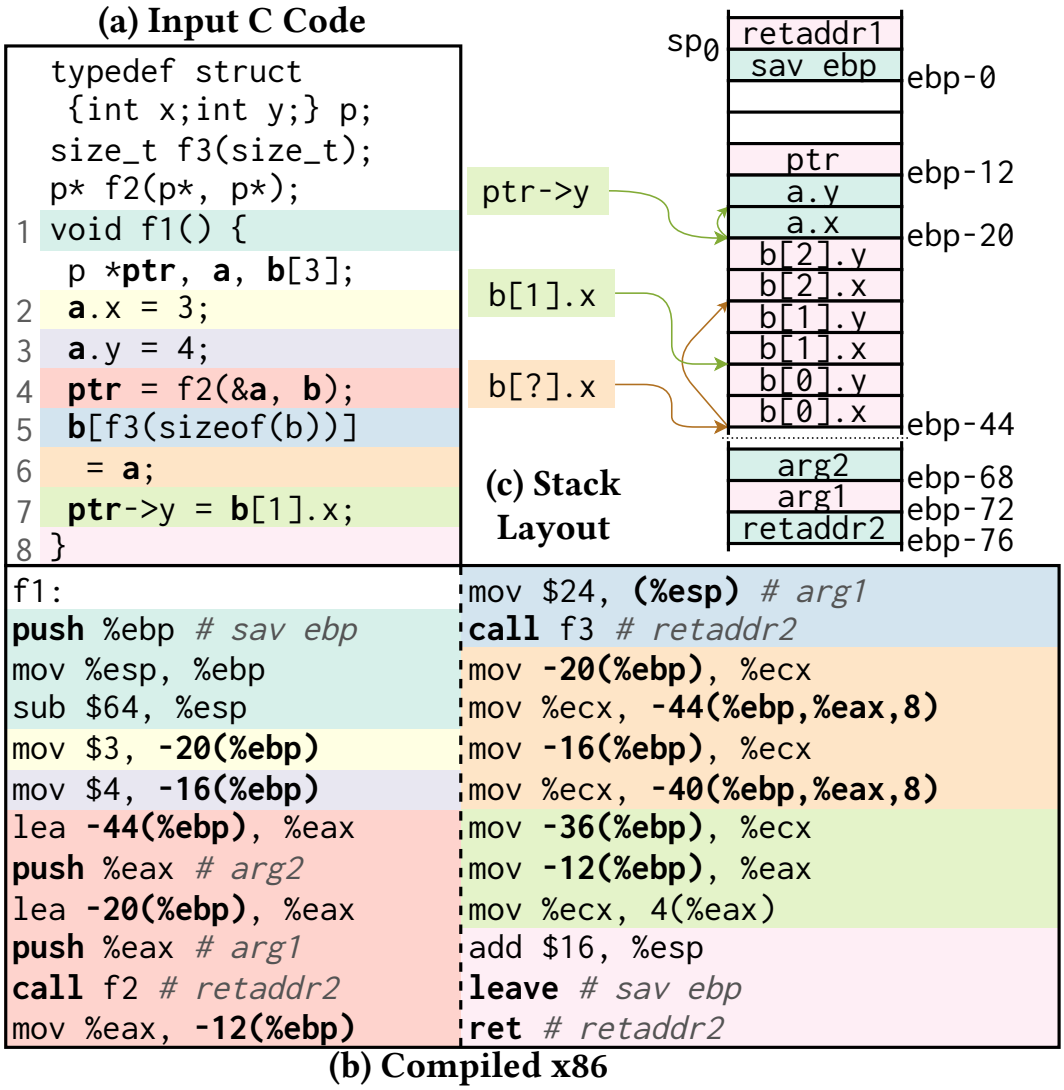


Figure 5.1: An example function and its stack frame. `f2` returns one of its arguments. `f3` returns a value less than its first argument. Bold values in (a) and (b) refer to the stack. For (c), assume `f2` returned `&a`, and `f3` returned 2.

5.1.2 Stack Symbolization

Stack symbolization is the process of labeling direct references to the stack with symbols that denote distinct local variables. Direct references to local 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 7 in Figure 5.1. The function computes the address to this

element using the value of `ebp` as base. The `ebp` register itself holds the value $\texttt{sp}_0\texttt{-4}$. Hence, the pointer computed by this instruction can be expressed as $\texttt{sp}_0\texttt{-4-36}$. 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 $\texttt{sp}_0\texttt{-4-44}$ 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 $\hat{\texttt{b}}\texttt{+8}$.

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 `2` and `3`, are folded into direct offsets to the frame pointer in optimized binaries and do not reveal their underlying structure. For example, in order to determine the bounds of variable `a`, an analysis has to establish that the access through `ptr` to `ebp-20` in line `7` can refer to the `8` byte memory area allocated to `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 `5` 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 Chapter 2, 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 [59]. 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, in rare cases, outside its corresponding array.

5.2 Design and Implementation

Figure 5.2 illustrates WYTIWYG's binary recompilation process in two phases. First, we rely on BinRec [12] to recover the target binary's CFG. BinRec uses a binary tracer (S2E [34]) 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 2.4. This program can already be recompiled, but it lacks variable information.

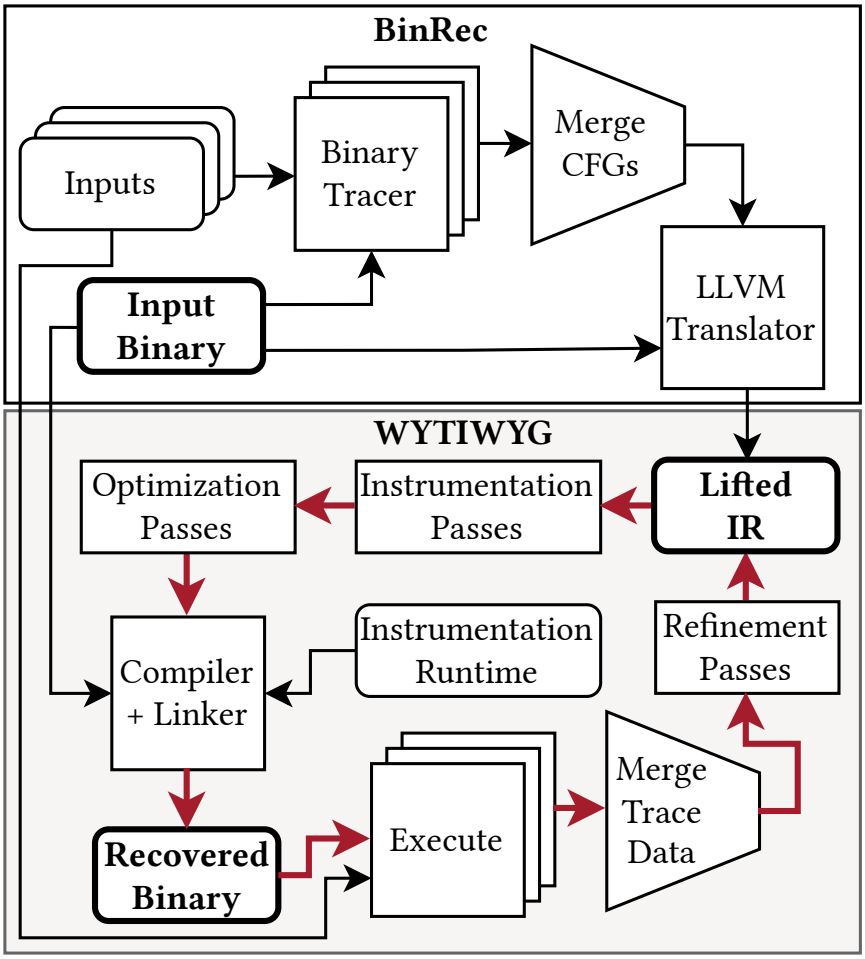


Figure 5.2: Overview of WYTIWYG. The upper section corresponds to the original BinRec recompiler. The lower section outlines our contribution. The red highlighted transitions correspond to the *Refinement Lifting* process.

We split the symbolization process into multiple steps with dedicated dynamic analyses and IR-

level transformations, as shown in Figure 5.2. This is similar to how certain passes are used to simplify and canonicalize the IR in regular compiler pipelines. In this phase, WYTIWYG iteratively symbolizes local variables in each function using dynamic analyses.

5.2.1 Dynamic Stack Symbolization

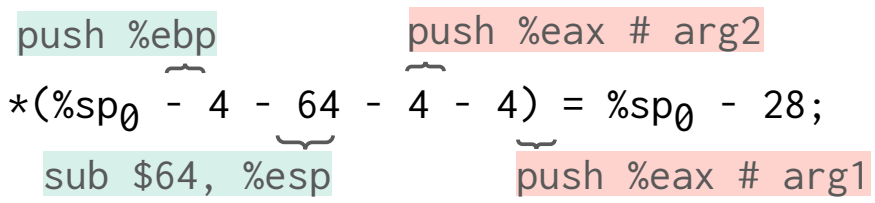
To symbolize local variables, WYTIWYG employs two refinements. The first identifies all direct stack references and rewrites them into expressions relative to $\texttt{sp}_0$. The second then determines the maximal offsets of pointers derived from each direct stack reference, uses this information to compute a stack layout for each function, and labels all direct references with symbols referring to variables within the stack layout.

5.2.2 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 5.1.2, direct stack references are pointers within a function that are computed as a sequence of constant displacements to $\texttt{sp}_0$. By folding all uses of $\texttt{sp}_0$, we can identify all direct stack references, and simplify them by replacing them with expressions of the form $\texttt{sp}_0\texttt{+offset}$. Consider the `push` instruction corresponding to `arg1` of the call in line `4` in Figure 5.1. It is initially lifted to 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:



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 `5`, could modify any value stored on the stack and therefore complicate determining whether `ebp` is a saved register (refer to Section 5.1.2).

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 [80]). 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 2.3.4). 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 Figure 5.1b, the register `edx` is not used once. 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 also need to make it an argument to `f1`. 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. We consider each forwarded register as saved, unless we classify it as an argument in any of the functions it is passed to.

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 $\texttt{sp}_0$, all direct references to objects on the emulated stack are expressed in terms of $\texttt{sp}_0$. These rewritten references serve as "base pointers" to local variables in the next refinement.

5.2.3 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 $\texttt{sp}_0$ 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 `6` and `7` of Figure 5.1. 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

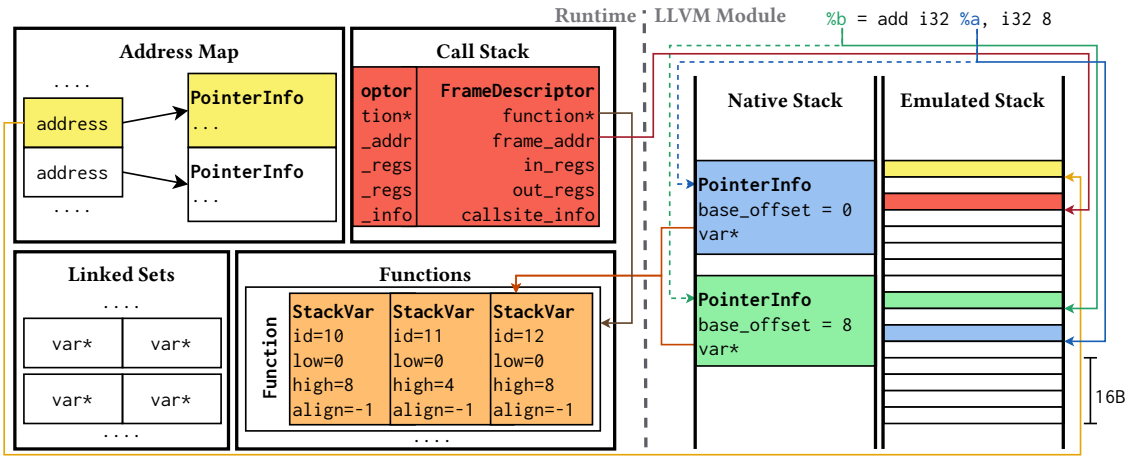


Figure 5.3: Overview of our tracing runtime.

object.

This also means that if `f3` 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.

5.2.4 Runtime Overview

To track direct and indirect stack references, we employ a runtime, which is illustrated in Figure 5.3. 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.

5.3 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 [34]. Rather than translating the machine code to LLVM IR while the program is running, we use a modified version of RevGen [33] 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.

5.3.1 Function Recovery

We implemented a function recovery similar to the approach detailed in Nucleus [15].

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.

5.3.2 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.

5.4 Evaluation

To evaluate our prototype, we first examine whether our approach retains the functionality of the original binary. We then measure the performance of symbolized binaries to assess whether our approach improves LLVM's ability to reoptimize lifted binaries. Finally, we compare the recovered stack-layouts with the ground-truth provided by the compiler to quantify the accuracy that can be achieved using our approach.

We target the SPECint 2006 benchmark suite, which has been widely used in previous binary lifting and recompilation literature [12, 14, 51, 75]. This benchmark-suite comprises 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 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 [12], McSema [41], RetDec [66] and mctoll [115] as the best available binary lifters targeting a compiler-level IR [75]. 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 [49]. 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 symbolization of stack variables.

5.4.1 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 would explore different paths in the lifted binary. We used the additive lifting approach described in Section 3.2.2 to generate sufficient coverage for these binaries. 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 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 disas-

Table 5.1: Normalized runtime of recompiled binaries relative to the runtime of their respective input binary for each configuration (**SW** = SecondWrite).

		BinRec / CN				**SW**
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

sembler. `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 wytiwyg/tables. For example, the jump table in the PIC version of the function `BZ2_decompress` from the binary `bzip2` was entirely missing. Even

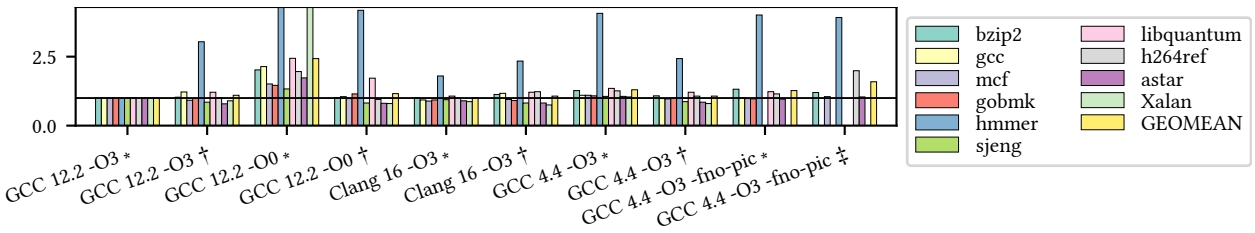


Figure 5.4: Normalized runtime of input (∗) binaries, binaries recompiled and symbolized with WYTIWYG (†), and binaries recompiled and symbolized with SecondWrite (‡) relative to the runtime of the respective binaries compiled and optimized with GCC 12.2.

when enabling speculative disassembly, the jump-targets were not present in the lifted LLVM IR. We assume that SecondWrite cannot identify speculative control transfer targets if the references to them are not encoded as absolute addresses in the binary's data.

5.4.2 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 multithreading 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 5.1 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 consequently, allows for full-scale program reoptimization.

We also improved the non-symbolized baseline for unoptimized binaries compared to the original version of BinRec from $0.98x$ [12] 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 particularly 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 5.4 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 [12, 14]. However, Figure 5.4 reveals, that more recent compilers are able to drastically reduce the runtime of this benchmark, to where 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 [14]. 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 that 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 that 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.

5.5 Discussion and Limitations

5.5.1 Binary Compatibility

Naturally, WYTIWYG can only recover local variables if we can identify functions that have regular stack-frames. Our implementation requires functions to have exactly one entry point, and control transfers between functions to be implemented using `call`- and `ret`-instructions (except for tail calls). The programs are also required to use the stack pointer register and have

it point to the bottom of the stack.

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 [121], to hide data-flow of pointers.

5.5.2 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 leads to 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` [99], 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 5.1.2, 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.

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 program retains the correct functionality for traced inputs. If a new input exhibits invalid behaviours in the recompiled binary, the program 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.

5.6 Conclusion

With WYTIWYG, we implement a novel, and fully automated, dynamic analysis based approach that recovers function local variables within lifted binaries. We demonstrate that moving stack objects from the emulated to the native stack this way enables us to significantly reoptimize legacy binaries.

Chapter 6

Discussion and Future Work

Chapters 3, 4, 5 detail our contributions in improving the state-of-the-art in binary recompilation. However, a fundamental question that rises amongst the users of recompilation is - "what guarantees do we get as part of the recompilation process?" For instance, if the recompiled binaries are to be deployed with critical security fixes, it is necessary to reason about their security and correctness.

In this chapter, we discuss certain semantic and correctness differences in the recompiled output that may result due to the design decisions made by recompilers and the underlying compiler infrastructure that they rely on. We show that well studied source-level miscompilation failures, such as those relating to code removal due to aggressive optimizations, also translate to recompilation. Then, we discuss how memory corruption-based exploit primitives in the input binary manifest in the recompiled binary. Finally, we provide an overview of the planned future work in the areas of cross-ISA recompilation.

6.1 Compiler-Induced Correctness Gaps

Chapter 2 demonstrates that precise and complete recovery of semantics from binary code is a hard problem. Moreover, Section 3.1.2 shows that there is an inherent difference in the abstractions provided by the hardware and that available as part of the lifted IR. This makes it difficult to precisely map the exact semantics of the various individual hardware instructions to LLVM IR. Previous work that attempts to verify binary lifters using *translation validation* [37], leverages formal semantics defined for x86/x64 hardware instructions and for LLVM IR. The core idea is to first perform symbolic verification of the translation of individual instructions to LLVM IR. This validation approach is then scaled to the full-program level, after a subset of IR-based optimizations are applied.

Verification failures at the instruction level suggest bugs in the machine code to LLVM IR translator. On the other hand, full-program verification failures suggest that certain optimizing transforms may not be semantics preserving. However, we note that verifying the *translation* of machine code to LLVM IR, i.e lifting, may not be enough to ensure correct recompilation. Section 2.4.2 emphasizes the differences between lifting and recompilation. We elaborate on some key concerns along this direction.

6.1.1 Code Removal

The removal of critical code due to IR-level optimizations may lead to changes in program behavior that impact correctness and security guarantees [113, 116]. For instance, consider code as shown in Listing 6.1 that scrubs sensitive buffer memory to prevent information leaks. Since the result of the `memset` call is not used, aggressive compiler optimizations may mark such code as "dead", and remove it.

```
/* Commit: d4c5efdb97773f59a2b711754ca0953f24516739 */
// drivers/char/random.c
static ssize_t extract_entropy(struct entropy_store *r, void *buf,
                                                size_t nbytes, int min, int reserved)
    // ...
    /* Wipe data just returned from memory */
    memset(tmp, 0, sizeof(tmp));
    // + memzero_explicit(tmp, sizeof(tmp));
}

/* Implementation of memzero_explicit */
// include/linux/string.h
static inline void memzero_explicit(void *s, size_t count)
{
        memset(s, 0, count);
        barrier_data(s);
}

// include/linux/compiler.h
# define barrier_data(ptr) __asm__ __volatile__("": :"r"(ptr) :"memory")
```

Listing 6.1: Call to `memset` that scrubs sensitive data from memory, and kernel implementation of `memzero_explicit` that uses compiler barriers.

In fact, the Linux kernel had to introduce a new routine `memzero_explicit` that uses compiler barriers to ensure that the call to `memset` does not get optimized away. But, the compiler barriers inserted to convey this programmer intent to the compiler do not generate any binary-level artifacts. As a result, when the binary is lifted to IR for the purpose of recompilation, this crucial information has already been lost. Off-the-shelf optimizations that are applied as part of IR refinement may consider the call to `memset` as dead code and eliminate it, unintentionally reintroducing information leaks that the original binary was protected against.

```
// Implementation of memset_explicit from LLVM's libc
// https://github.com/llvm/llvm-project/blob/main/libc/src/string/memset_explicit.cpp
[[gnu::noinline]] LLVM_LIBC_FUNCTION(void *, memset_explicit,
                                     (void *dst, int value, size_t count)) {
  // Use the inline memset function to set the memory.
  inline_memset(dst, static_cast<uint8_t>(value), count);
  // avoid dead store elimination
  // The asm itself should also be sufficient to behave as a compiler barrier.
  asm("" : : "r"(dst) : "memory");
  return dst;
}
```

Listing 6.2: LLVM libc implementation of `memset_explicit`, added as part of the C23 standard

Previous work details multiple remedies against the problem of harmful dead store elimination [116] in compilers. However, most of them fail to translate to recompilation (in the context of Linux-based operating systems) -

- Reliance on platform-provided functions : Listing 6.2 details the implementation of `memset_explicit` as added to LLVM's libc as part of the C23 standard. Note that, the underlying implementation uses an inline `memset` call, and pairs it with a compiler barrier to ensure that it does not get optimized away. However, the recompiler is only going to observe the call to `memset`, as no hardware instruction is emitted for the compiler barrier, which leads us back to the original problem.
- Disabling optimizations : Section 2.4 shows how emulation-based lifting introduces significant overheads. Recompilers rely on aggressive off-the-shelf compiler optimizations to remove redundant register and flag computation, improve inlining, etc. to improve performance of the recompiled output. Doing away with optimizations will be detrimental to the code size and runtime performance of the output.
- Hiding semantics : Programmers may hide the semantics of the scrubbing `memset` operation

from the compiler to prevent it from identifying the code as dead. For instance, the function may be defined as part of a different compilation unit, invoked through a `volatile` function pointer or by using inline assembly snippets. When the binary is lifted to IR, we lose the abstraction boundary of distinct compilation units. The IR can be considered to similar to what is seen when Link-Time Optimization (LTO) is applied, which completely defeats the original goal of hiding semantics. Moreover, looking at the binary, it is impossible to know whether the `volatile` attribute was applied to a function or not. Inline assembly snippets are treated similar to compiler-generated machine code during lifting, as it can be impossible to differentiate between the two in the binary.

- Forcing memory writes : Although memory barriers and volatile data pointers may not work, using complicated computations that surive compiler optimizations in the binary-to-binary round trip may prevent dead store elimination.

Another possible solution is to implement an analysis pass that identifies "sensitive" functions through user-provided annotations or heuristics and marks them as `no_optimize`. As the size of such sensitive code is usually quite low, the performance penalties may be manageable.

6.1.2 Lifting Exact Semantics

Certain x86/x64 instructions impose side-effects that can be difficult to model at the LLVM IR level. Consider the example of the `movntq` instruction. This instruction bypasses the cache hierarchy and directly writes the value to main memory to minimize cache pollution. However, analysis of modern lifters (e.g. Remill) suggest that they handle this instruction similar to an ordinary store. Another example is that of `prefetchntq`, which is used to prefetch data into the non-temporal cache structure and into a location close to the processor to minimize cache pollution. Incomplete modeling of this instruction forced Dasgupta et al. to exclude 6 programs from their evaluation [37]. They also do not handle system-level instructions such as those re-

lating to operating systems (e.g. `cpuid`), protection levels, I/O, concurrency, etc. Though this may not lead to severe security implications, the examples demonstrate that verifying semantic equivalence between LLVM IR and the underlying hardware instruction may not be always possible.

Recent work has shown that microarchitectural side-channels are a major privacy threat [104]. An attacker can learn minute details about a running victim program by monitoring its hardware resource utilization (such as the cache), precisely observing timing differences across different inputs or tracking power draw [60]. We discuss timing side-channels for the purpose of brevity. A timing side-channel may leak information about which code paths are exercised in a program depending on the time it takes to execute certain inputs. To thwart such attacks, programmers may adopt "constant-time" programming techniques for secret sensitive parts of the code [31]. Here, divergent control flows are written such that they clock the same number of cycles or execute in the same time regardless of which branch is taken. "Timing safe" instructions, that do not leak any information about the operands, are encouraged to be used. The generated machine code captures the programmer intent informally, within the chosen hardware instructions and the control flow. Note that, binary-level analyses cannot reliably infer if a certain function is timing sensitive as no accompanying metadata can be extracted. Depending on how the lifter chooses to translate individual instructions in the diverging branches and the impact of the downstream optimizations, it is difficult for the recompiler to precisely ensure that the timing characteristics are maintained. Even if instruction *translation* can be verified to be "safe", the compiler may reorder instructions around introducing new side-channels that did not exist earlier.

We envision recompilers in the future that enable "selective" recompilation, such as restricting translation to certain call trees of the binary while keeping the rest of it intact. This helps rid the overheads due to instruction emulation for parts of the binary that do not need to be rewritten. For downstream applications such as patching that deal with modifications at the granularity of individual functions or instructions, support for partial recompilation may be useful. Such

an ability also enables resolving the problem of recompiling timing-sensitive functions by preserving them "as is" in the recompiled binary. The recompiler will have to sink the values of emulated input registers before making a call to such a function, and add reloads for values of the corresponding output registers.

An alternate approach is to use a microarchitectural timing analyzer such as llvm-mca [1] to estimate precise timing differences between the patched and the unpatched versions of the binary [57]. Recompilers may implement a flag that forces similar timing characteristics on the recompiled output per the original binary. However, this will require tuning instruction selection heuristics in the compiler to honor these constraints, leading to a tradeoff between performance and timing equivalence.

6.2 Out-of-bounds Exploits

A central application of recompilers is modernizing legacy binaries. However, legacy binaries are rife with out-of-bounds vulnerabilities such as buffer and heap overflows. In this section, we discuss the transformation of exploit primitives, that are a part of the original binary, on lifting and how they manifest as part of the recompiled output.

6.2.1 Stack Management

Stack-based buffer overwrites are powerful exploit primitives. In the absence of a stack canary, an attacker could overwrite the return address of the function and hijack control flow. Or, depending on the layout of the stack they could also mount a data-oriented attack by overwriting critical control variables stored on the function-local stack frame. Overreads, on the other hand, can leak crucial variable information which can be combined with other primitives to mount more reliable attacks.

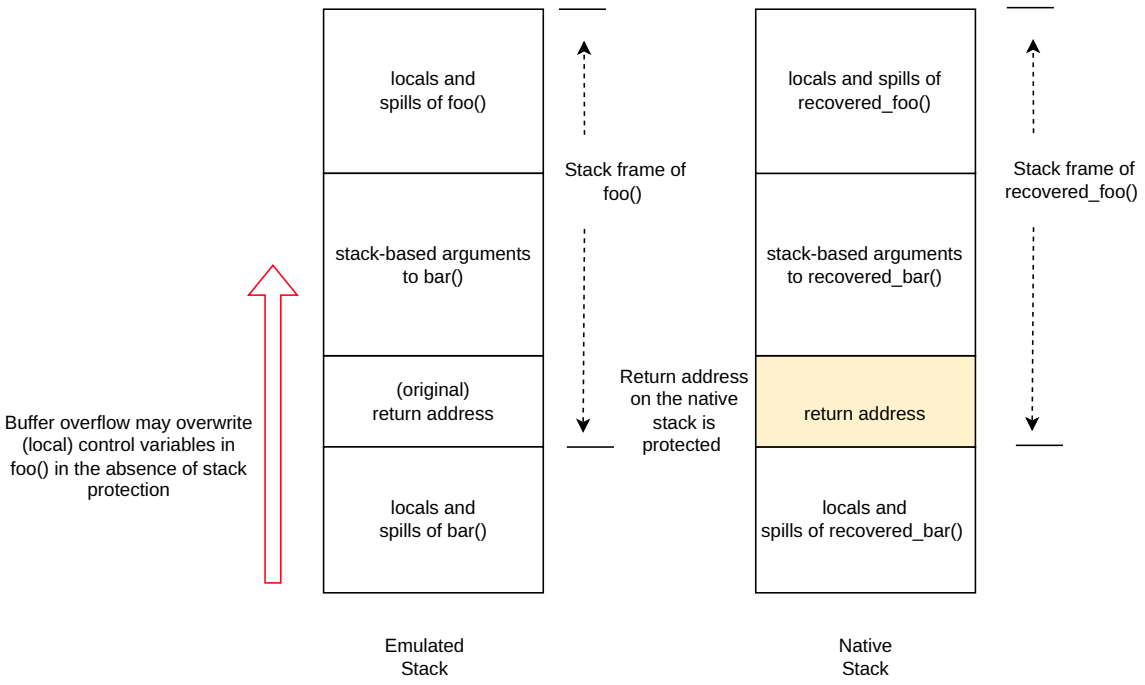


Figure 6.1: Buffer overflow in the recompiled binary with an emulated stack.

The stack can be lifted in roughly 3 levels of granularity -

- Emulated stack : A global byte array is allocated that models the entire program stack of original binary (eg. McSema, BinRec, Rev.Ng, Polynima).
- Lifted stack : After inferring bounds on the maximum frame size, the recompiler allocates a per-function byte array that represents the stack frame for the particular function (eg. mctoll).
- Lifted objects : Individual objects, that roughly represent program variables in the original binary, are identified and they are moved from the emulated stack to the native stack (eg. SecondWrite, WYTIWYG).

In the context of recompilation, stack-based out-of-bounds primitives manifest in different ways depending on how the stack is handled.

Consider the case of the emulated stack as shown in Figure 6.1. To emulate the `call` instruction, the recompiler pushes the original return address value on the emulated stack. However, if

functions are recovered as part of the IR, the value pushed to the native stack is the one used to return to the caller. So, buffer overwrites that aim to hijack control flow by controlling the return address are thwarted by design. But, notice from Figure 6.1, that the overwrite may actually be more powerful in certain cases.

Figure 6.2 highlights the case with a lifted stack. Depending on how the individual objects are laid out, the original buffer overflow primitive may become even more powerful and allow overwriting control variables of the emulator as well as the return address. These issues become all the more complicated with lifted objects and under the influence of aggressive compiler optimizations such as inlining.

6.2.2 Heap Management

Temporal memory safety errors such as double-free, and use-after-free are leading weaknesses in modern software [4]. Exploits may also attempt to use heap-based buffer overflows to perform control flow hijacks or non-control data attacks by predicting storage reuse or other strategies used by the allocator.

For non-static binaries, heap management is typically performed through APIs provided by an external library such as glibc. Therefore, heap management operations (such as those relating to allocation and freeing of memory) manifest as external library calls and are lifted as such. Examples are calls to functions such as `malloc/free` in C, and `new/delete` in C++. As recompilers often do not implement explicit tracking of heap objects, heap exploit primitives persist as is.

Recompilers may choose to symbolize heap pointers and implement a pass that identifies possible heap overflows at the IR level. It is also possible to link in a safe allocator library that prevents out-of-bounds accesses at the cost of performance.

The above problems point to a much larger issue of the inherent gap in recompilation to

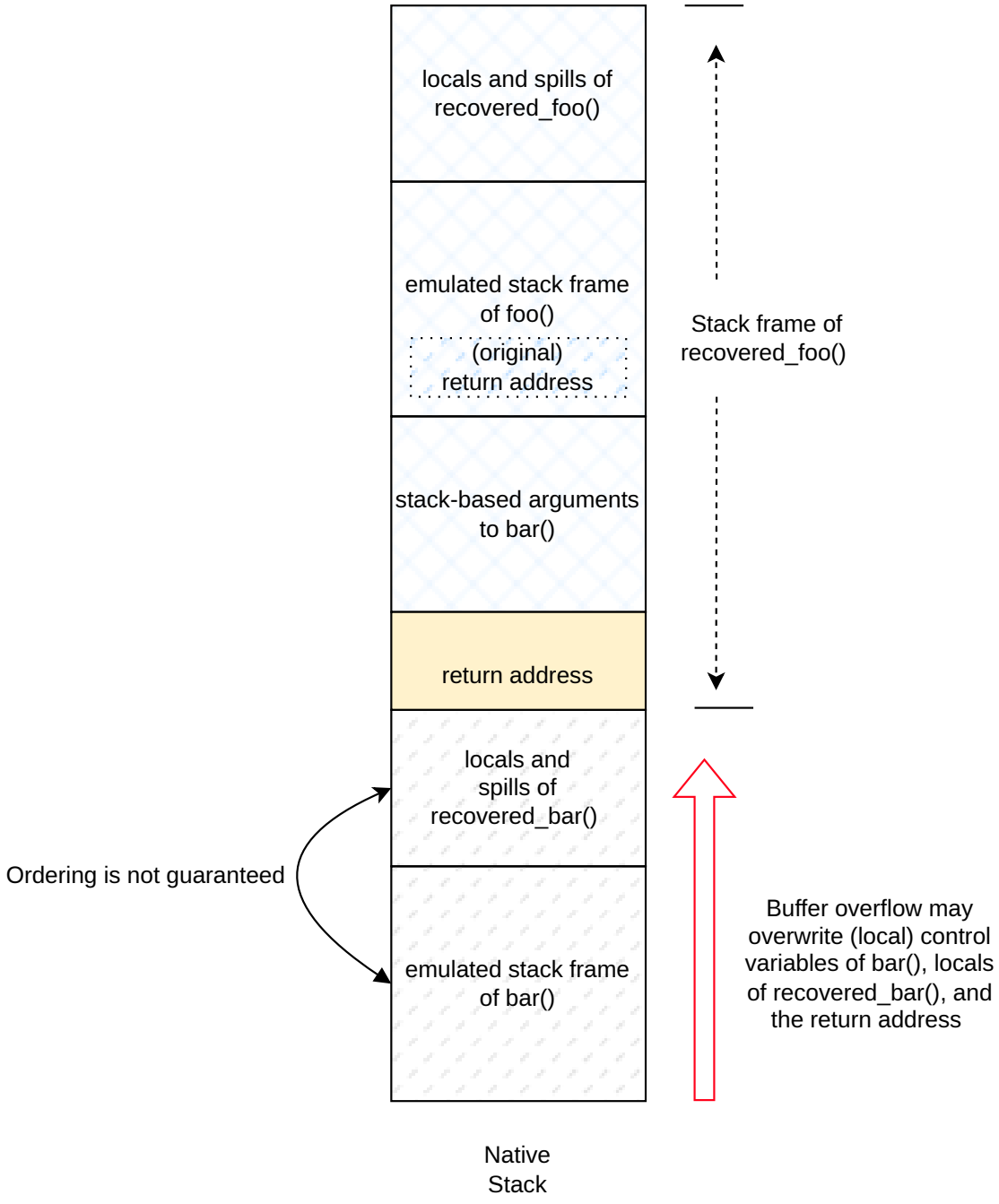


Figure 6.2: Buffer overflow in the recompiled binary with lifted stack.

maintain security and correctness guarantees with respect to the original binary while also focusing on optimizations.

6.3 Extensibility

6.3.1 Integrating Polynima and WYTIWYG

The two major pieces of our recompilation work, Polynima and WYTIWYG, are developed as parts of the same high-level project structure which allows us to leverage solutions to shared sub-problems. In the future, we aim to extend our stack symbolization approach to better optimize legacy multithreaded binaries. Fundamentally, we believe that our approach to use instrumentation to recover bounds of stack variables and lift them in the IR can reliably work with multithreaded binary programs. We are currently extending the tracing runtime to implement guards that synchronize concurrent accesses to shared structures. Note that the additive lifting runtime already implements a version of this, as we ensure that only a single new control flow is recorded across different threads.

6.3.2 Cross-ISA Recompilation

A significant appeal of developing techniques that improve the state-of-the-art in recompilation is to enable the cross-ISA translation of binaries. Newer processor architectures, and therefore, the corresponding ISAs such as ARM and RISC-V, are gaining mainstream adoption for building desktop and server compute. Even though source languages such as C and C++ are mostly hardware agnostic, performance critical applications often make use of target specific intrinsics or inline assembly. So, modifying large source projects to reliably compile for a different ISA can be a major hassle in terms of the time and the effort involved. In such a case, binary recompilation offers an alternative path to support legacy programs for newer architectures as we can reuse much of the existing compiler infrastructure (in this case LLVM) to ***retarget*** the lifted IR to compile for another ISA.

A naive approach to support cross-recompilation is to update the data-layout of the LLVM IR module to reflect that of the target architecture. However, care must be taken to ensure that the IR is amenable to such a transformation. For instance, inline assembly written in x86/x64, which we use for stack-switching, will not be compatible with a different target ISA. Therefore, we need to know signatures of all external library calls. More generally, we need to eliminate all instances of inline assembly and the use hardware specific intrinsics (if any) in the lifted IR.

Care must be taken to ensure that pointers and structures that are used to interface with external libraries are correctly laid out in memory. For instance, certain libc functions may have different ordering of arguments depending on the underlying ISA. Moreover, translating for an architecture with a different pointer width, such as from x86 (32 bit) to AArch64, requires precisely identfying and extending all pointers which is a hard problem. The current approach we use for handling unknown callbacks, i.e. inserting trampolines at original function addresses, will also fail to work. As a result, it is imperative to apply refinements to the lifted IR and recover as much information as possible (either statically or dynamically) before translating to a different ISA.

Our recompiler currently implements basic support for cross-recompilation of x86 binaries to ARM32 and x64 binaries to AArch64. We have tested it on the mcf binary from the SPECint 2006 benchmark suite.

Chapter 7

Conclusion

Binary recompilation holds the promise of making available the rich analysis and transformation ecosystem of the compiler, that is typically used with source-level programs, to modify and re-optimize binary programs. But, state-of-the-art tools in this space have not seen much adoption due to practical concerns.

Current recompilers are entirely static or dynamic in their approach to recompilation. Static tools are fast, but rely on heuristics for reasoning about possible control flow paths in the program and recovering stack-local variable information. Transformations that rely on unsound analyses may lead to divergences from the original program behavior or even faults during recompiled binary execution. Dynamic techniques, on the other hand, offer precision but are inefficient due to their high tracing overheads. Crucially, none of these tools generally handle multithreaded binaries, that are ubiquitous in the modern software space.

In this dissertation, we demonstrated improvements to the state-of-the-art in binary recompilation across two complementary axes. We presented Polynima [39], the first practical recompiler that is able to reliably lift and recompile complex real-world multithreaded binaries and benchmark suites. Polynima implements a novel hybrid control flow recovery strategy that com-

bines the benefits of static and dynamic lifters, along with soundly handling the inherent non-determinism in multithreaded programs. Finally, we designed a dynamic procedure to detect implicit synchronization primitives in binaries and used that to improve overheads induced due to the conservative handling of shared memory accesses in the lifted IR.

Then, we tackled the problem of inferring stack layouts in binary programs through two novel approaches: StackBERT [38] and WYTIWYG [90]. StackBERT is a learning-based technique to recover function frame sizes in binary programs. We showed the inadequacy of current static approaches to reason about the program stack in binaries and provided evidence that our trained models are valuable for target architectures that are largely unsupported by existing analysis tools. WYTIWYG significantly improved upon this by designing instrumentation-based dynamic analyses that recover program variables. Our fine-grained approach tracks the dataflow across stack memory operations at runtime and infers bounds that helps delineate distinct objects. By using this information to refine the lifted IR, we showed that WYTIWYG makes it possible to leverage the full potential of the compiler ecosystem to recompile and reoptimize legacy binaries.

Bibliography

[1] Llvm mca. `https://llvm.org/docs/CommandGuide/llvm-mca.html`. Accessed: 2021-06-28.

[2] IDA Pro, 1991.

[3] Tool interface standard (tis) executable and linking format specification (elf), 1995.

[4] 2023 cwe top 10 kev weaknesses, 2023.

[5] CVE-2023-24042, 2023.

[6] Literal pools, 2024.

[7] Pe format, 2024.

[8] M. Adler. pigz, 2007.

[9] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 70–83, New York, NY, USA, 2019. Association for Computing Machinery.

[10] T. Ahmed, P. Devanbu, and A. A. Sawant. Finding inlined functions in optimized binaries. 2021.

[11] S. Al Bahra. ConcurrencyKit, 2011.

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

[13] S. Alvarez. Radare2: Libre reversing framework for Unix geeks, 2006.

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

[15] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189, New York, NY, USA, 2017. IEEE.

[16] Apple. About the rosetta translation environment. `https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment`, 2024.

[17] G. Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):1–84, 2010.

[18] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, 2014.

[19] T. Bastian, S. Kell, and F. Zappa Nardelli. Reliable and fast dwarf-based stack unwinding. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, volume 3, pages 1–24. ACM New York, NY, USA, 2019.

[20] E. Bauman, Z. Lin, K. W. Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[21] S. Beamer, K. Asanović, and D. Patterson. The GAP benchmark suite, 2017.

[22] M. Beck, K. Bhat, L. Stričević, G. Chen, D. Behrens, M. Fu, V. Vafeiadis, H. Chen, and H. Härtig. AtoMig: Automatically migrating millions lines of code from TSO to WMM. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 61–73, New York, NY, USA, 2023. Association for Computing Machinery.

[23] F. Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, 2005. USENIX Association.

[24] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[25] E. Bendersky. pyelftools: Parsing elf and dwarf in python. `https://github.com/eliben/pyelftools`, 2011.

[26] S. Bhattamishra, K. Ahuja, and N. Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

[27] E. Botcazou, C. Comar, and O. Hainque. Compile-time stack requirements analysis with gcc. In *Proceedings of the 2005 GCC Developer's Summit*, page 93. Citeseer, 2005.

[28] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[29] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[30] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[31] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 913–926, 2020.

[32] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[33] V. Chipounov and G. Candea. Enabling sophisticated analyses of ×86 binaries with RevGen. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 211–216, New York, NY, USA, 2011. IEEE.

[34] V. Chipounov, V. Kuznetsov, and G. 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*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. Association for Computing Machinery.

[35] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, 2017.

[36] D. D. I. F. Committee. Dwarf debugging information format version 5, 2017.

[37] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–671, 2020.

[38] C. Deshpande, D. Gens, and M. Franz. StackBERT: Machine learning assisted static stack frame size recovery on stripped and optimized binaries. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, AISec '21, pages 85–95, New York, NY, USA, 2021. Association for Computing Machinery.

[39] C. Deshpande, F. Parzefall, F. Hetzelt, and M. Franz. Polynima: Practical hybrid recompilation for multithreaded binaries. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1126–1141, 2024.

[40] A. Di Federico, M. Payer, and G. Agosta. Rev.Ng: A unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 131–141, New York, NY, USA, 2017. Association for Computing Machinery.

[41] A. Dinaburg and A. Ruef. Mcsema: Static translation of x86 instructions to llvm. Presented at *REcon 2014* (Montreal, Canada), 2014.

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

[43] T. Dullien and S. Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *Proceeding of CanSecWest*, pages 1–7, 2009.

[44] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 51–60, 2013.

[45] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[46] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, volume 32, pages 3708–3719, 2019.

[47] C. Fu, K. Yang, X. Chen, Y. Tian, and J. Zhao. N-bref: A high-fidelity decompiler exploiting programming structures. 2020.

[48] GCC. Gcc ipa split optimization pass, 2024.

[49] P. Goodman and A. Kumar. Lifting program binaries with mcsema. Presented at *9th International Summer School on Information Security and Protection* (Canberra, AU) *(ISSIP '18)*, 2018.

[50] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. *SIGARCH Comput. Archit. News*, 17(2):54–63, 1989-04.

[51] A. Gussoni, A. Federico, P. Fezzardi, and G. Agosta. Performance, correctness, exceptions: Pick three. In *Binary Analysis Research Workshop 2019*, BAR 2019, Reston, VA, USA, 2019. The Internet Society.

[52] M. Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.

[53] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[54] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991-01.

[55] hfiref0x. LightFTP.

[56] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[57] M.-Y. Hsu, F. Hetzelt, D. Gens, M. Maitland, and M. Franz. A highly scalable, hybrid, cross-platform timing analysis framework providing accurate differential throughput estimation via instruction-level tracing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 821–831, 2023.

[58] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2023-01.

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

[60] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi. {V0LTpwn}: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1445–1461, 2020.

[61] T. kernel development community. Orc unwinder. `https://www.kernel.org/doc/html/latest/x86/orc-unwinder.html`, 2021.

[62] S. H. Kim, C. Sun, D. Zeng, and G. Tan. Refining indirect call targets at the binary level. In *NDSS*, 2021.

[63] N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[64] M. Kolsek and the 0patch Team. Did microsoft just manually patch their equation editor executable? why yes, yes they did. (cve-2017-11882). `https://blog.0patch.com/2017/11/did-microsoft-just-manually-patch-their.html`, 2017.

[65] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 54–64, 2007.

[66] J. Křoustek, P. Matula, and P. Zemek. Retdec: An open-source machine-code decompiler. Presented at *Botconf 2017* (Montpellier, France), 2017.

[67] J. Křoustek, P. Matula, and P. Zemek. RetDec: An open-source machine-code decompiler. 2017.

[68] M. Larabel. The linux kernel is now vla-free: A win for security, less overhead and better for clang. `https://www.phoronix.com/scan.php?page=news\_item\&px=Linux-Kills-The-VLA`, 2012.

[69] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, New York, NY, USA, 2004. IEEE.

[70] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[71] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium 2011*, NDSS 2011, Reston, VA, USA, 2011. The Internet Society.

[72] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium 2010*, NDSS 2010, Reston, VA, USA, 2010. The Internet Society.

[73] C. Linn, S. Debray, G. Andrews, and B. Schwarz. Stack analysis of x86 executables. *Manuscript. April*, 2004.

[74] Z. Liu, Y. Yuan, S. Wang, and Y. Bao. SoK: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1100–1119. IEEE, 2022.

[75] Z. Liu, Y. Yuan, S. Wang, and Y. Bao. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1100–1119, New York, NY, USA, 2022. IEEE.

[76] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 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.

[77] S. Lyubka. Mongoose, 2004.

[78] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, pages 165–171. IEEE, 1994.

[79] D. Maier and F. Toepfer. Bsod: Binary-only scalable fuzzing of device drivers. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 48–61, 2021.

[80] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. *System V Application Binary Interface — AMD64 Architecture Processor Supplement (Draft Version 0.99.6)*, 2013.

[81] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[82] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pages 4505–4515. PMLR, 2019.

[83] X. Meng and B. P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[84] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–365, 2021.

[85] NSA. Ghidra reverse engineering framework, 2024.

[86] C. OS. Stack size analyzer. `https://www.chromium.org/chromium-os/ec-development/stack-size-analyzer`, 2017.

[87] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.

[88] M. Panchenko, R. Auler, B. Nell, and G. Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[89] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu. 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)*, pages 833–851. IEEE, 2021.

[90] F. Parzefall, C. Deshpande, F. Hetzelt, and M. Franz. What you trace is what you get: dynamic stack-layout recovery for binary recompilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1250–1263, 2024.

[91] J. Patrick-Evans, L. Cavallaro, and J. Kinder. Probabilistic naming of functions in stripped binaries. In *Annual Computer Security Applications Conference*, pages 373–385, 2020.

[92] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 2021 16th Joint Meeting on Foundations of Software Engineering*. ACM, 2021.

[93] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana. Xda: Accurate, robust disassembly with transfer learning. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[94] S. Poeplau and A. Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *NDSS 2021, Network and Distributed System Security Symposium*. Internet Society, 2021.

[95] J. Poimboeuf. objtool: add tool to perform compile-time stack metadata validation. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=442f04c34a1a467759d024a1d2c1df0f744dcb06`, 2016.

[96] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. Razor: a framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1733–1750, USA, 2019. USENIX Association.

[97] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE, 2007.

[98] R. C. O. Rocha, D. Sprokholt, M. Fink, R. Gouicem, T. Spink, S. Chakraborty, and P. Bhatotia. 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*, PLDI 2022, pages 888–902, New York, NY, USA, 2022. Association for Computing Machinery.

[99] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, Berkeley, CA, USA, 2012. USENIX Association.

[100] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, Washington, D.C., August 2015. USENIX Association.

[101] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[102] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium 2011*, NDSS 2011, Reston, VA, USA, 2011. The Internet Society.

[103] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[104] J. Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, 2019.

[105] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 143–154, 2008.

[106] L. Torvalds. Re: [rfc 0/5] kernel: backtrace unwind support. `https://lkml.org/lkml/2012/2/10/356`, 2012.

[107] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[108] Valgrind. Vex ir, 2024.

[109] Vector35. Binaryninja, 2024.

[110] A. Vorobey and B. Fitzpatrick. Memcached, 2003.

[111] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. From hack to elaborate technique–a survey on binary rewriting. *ACM Comput. Surv.*, 52(3), June 2019.

[112] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 133–147, New York, NY, USA, 2020. Association for Computing Machinery.

[113] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao. Silent bugs matter: A study of {Compiler-Introduced} security bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3655–3672, 2023.

[114] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[115] S. B. Yadavalli and A. Smith. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, pages 213–218, New York, NY, USA, 2019. Association for Computing Machinery.

[116] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko. Dead store elimination (still) considered harmful. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1025–1040, 2017.

[117] Y. Yu, X. Qin, and S. Gan. Hdbfuzzer–target-oriented hybrid directed binary fuzzer. In *Proceedings of the 5th International Conference on Computer Science and Application Engineering*, pages 1–8, 2021.

[118] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, et al. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.

[119] M. Zalewski. American fuzzy lop, 2017.

[120] A. Zhou, C. Ye, H. Huang, Y. Cai, and C. Zhang. Plankton: Reconciling binary code and debug information. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 912–928, New York, NY, USA, 2024. Association for Computing Machinery.

[121] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In S. Kim, M. Yung, and H.-W. Lee, editors, *Information Security Applications*, pages 61–75, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[122] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.