Toward a Science of Software Reverse Engineering

by

Zion Leonahenahe Basque

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved March 2026 by the
Graduate Supervisory Committee:

Ruoyu (Fish) Wang, Co-Chair
Yan Shoshitaishvili, Co-Chair
Adam Doupé
Cristina Cifuentes

ARIZONA STATE UNIVERSITY

May 2026

ABSTRACT

Modern software systems are increasingly built, modified, and deployed by people who did not create them. As a result, software reverse engineering (SRE), the process of recovering structure, intent, and behavior from existing artifacts, has become a fundamental part of interacting with software. Despite its importance, SRE remains largely an art: its goals are inconsistently defined, its processes are poorly characterized, and its success is often measured only through downstream outcomes.

This dissertation argues that SRE can be transformed into a science by systematically studying three interconnected aspects of the discipline: the techniques that enable reverse engineering, the processes by which those techniques compose into effective workflows, and the applications that ultimately define reverse engineering's utility.

First, we study reverse engineering techniques by introducing closeness-to-source as a principled metric for evaluating decompilation quality and designing SAILR, a compiler-aware decompilation framework that demonstrates how metrics-driven design advances individual techniques. Second, we investigate the reverse engineering process through an empirical study of human and LLM-assisted workflows, demonstrating that understanding emerges through the iterative, strategy-dependent composition of multiple techniques. Third, we examine the applications of reverse engineering through PatcherY, an autonomous vulnerability patching system, showing that applications provide a concrete measure of whether our techniques and metrics serve real-world needs.

Together, these contributions provide a unified foundation for SRE that connects techniques, processes, and applications. By making each aspect explicit and measurable, this work takes a step toward a science of reverse engineering.

I dedicate this dissertation to Shayde, my cousin, my friend, and my reminder that life is short. I think about you often, imagining the life you would've lived, the children you would've raised, and how much fun it would have been to celebrate this event with you. I am older than you now. I promise I will live enough life for both of us, helping the world with my work, but also having fun, like I know you would've wanted me to.

I love you, Shayde.

ACKNOWLEDGMENTS

This dissertation is the culmination of the last five years of my life, bearing my name across its title page. Yet when I look back, I can only think of all the names that belong there alongside mine, people to whom I will be forever grateful. I am thankful to the countless individuals who paved the way for me to succeed, who pushed me forward when my strength ran low, and who inspired me to do something meaningful with my life.

First and foremost, I want to acknowledge my wife, Emma. You are my ride-or-die, my number-one fan, and the person I most want to impress in life. You have been my rock and my source of inspiration since we first started dating in high school. Through every deadline, stressful moment, and long talk, I am reminded how lucky I am to have a partner who makes the hard things feel manageable. You motivate me to push harder, but also give me the comfort of knowing that even if I fail, you will still have my back. Thank you for not letting me quit and for believing in me when I did not believe in myself. You are the greatest thing in my life. I love you.

I want to acknowledge my family, who supported me even as I journeyed far from home. To my mother, Priscilla, who had to be both mom and dad, you taught me how to work hard and how to carry a heart full of kindness for others. To my grandmother, Grace, who instilled in me a love for knowledge and inspired me to give my best in everything I do. To my sister, Ruby, whom I have always tried to emulate in moments of crisis and impress in moments of success. To my aunty, Davey, who taught me how to live life with a plan and be successful. To my father, better late than never! You taught me how to be funny and charismatic, something I benefit from every day of my life.

To my second family, my friends and colleagues, thank you. You shaped me into the researcher and person I am today. To Gage and Kirsten, thank you for supporting Emma and me throughout my entire PhD. You are lifelong friends and a constant source of joy in my life. Those months during COVID, just hanging out online, kept life fun. I will forever

cherish our Minecraft world.

To Wil, my best friend on this PhD adventure, you made this journey unforgettable. It was because of you that I even started this path. You taught me my first overflow. We have been through so much together, including paper submissions, international CTFs, conferences, career growth, and the chaos of two years of AIxCC. I will never forget the 4 a.m. crunches and the grind we endured to finish this degree. More than anything, I will remember that you were always a great friend.

To Ati Priya, your kindness carried me through difficult times, and your dedication to research pushed me to be better, especially during SURE. I am so glad we became friends, and I am excited to see the impact you will make in the future. To Jay, those lunches at Munchies taught me the importance of humor and how to laugh at the process. To Connor, from the early pwndevils days when we hacked through the night, you inspired me to sharpen my skills. I will always think of you as one of the most gifted and brilliant hackers I have ever met. And to the rest of Shellphish, you gave me a community I never knew I needed, a true family of hackers. I will continue to make you proud and carry forward the vision of hackademia.

Thank you, Adam, for your guidance and education. If Wil is the reason I started this journey, you are the reason I completed it. Sorry, Wil, Adam was just the better teacher. You taught me more than I can fully capture here, including negotiation, research, ethics, balance, charisma, and so much more. What I will remember most is your commitment to doing good science, grounded in real work, with no shortcuts.

To Yan, the coolest professor I have ever met, I always admired how fun you made research feel. Even under the most intense deadlines, you found a way to laugh. You taught me how to enjoy the process, even when the stakes are high. I hope you never forget the first question I asked in your class, “what is a graph?” Without your early guidance and inspiration, I might have taken a completely different path.

To Fish, my teacher, my mentor, and my ever-moving goalpost, thank you. It is hard to imagine how someone can walk into your life so casually, over sandwiches, and completely change your trajectory. You knew exactly when to introduce me to the field I would fall in love with, understanding programs. That moment changed the kind of hacker I became. It made me more curious, more driven, and more inspired than I ever expected to be. Through you, I became not just a hacker, but a builder, and that shift propelled everything that followed.

You were there at every step, pushing me to go deeper and do truly meaningful work. Even during the chaos of COVID, some of my biggest breakthroughs came at 2 a.m. over Zoom with you. You are relentlessly motivated by what interests you, not by grants or publication counts, and that shaped how I approach research. You are brutally honest and intensely focused, yet also unexpectedly kind. You are not only the best hacker I have ever met, you are exactly the mentor I needed.

I still remember one of my earliest moments of inspiration. After watching you solve four consecutive challenges first against thousands of hackers worldwide, we had a research meeting the following Monday. I told you, "One day, I will be as good as you at reversing and impress you, Fish." And you said, "I do not have a single doubt in my mind," even though I could barely solve easy challenges at the time. Those words meant more to me than you may ever know. Thank you. I will never be able to fully repay your mentorship, kindness, and belief in me.

Finally, to my ancestors, my kūpuna, for setting in motion the path I walk today. Our people have always been adventurers, storytellers, and warriors. It amazes me that the work I have found calls on all of those qualities you instilled. I will continue to share our story, of Hawaiians and our people, as I work to change the world. Your presence and history guide me forward.

I ka wā ma mua, i ka wā ma hope.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Throughout the history of computing, humans have sought to understand systems they did not create. In software, this process falls under *reverse engineering*: the act of recovering structure, intent, and behavior from existing artifacts. While traditionally associated with binary analysis, reverse engineering today extends far beyond compiled code. Modern software ecosystems increasingly involve code produced by others, poorly documented, or synthesized by automated systems such as large language models (LLMs). Developers and analysts now routinely understand and modify systems they did not author.

Software reverse engineering (SRE) underpins the security, reliability, and transparency of software systems. SRE supports vulnerability discovery, malware analysis, debugging, and system maintenance in environments where source code is unavailable or insufficient. As software development shifts toward reuse and automation, reverse engineering has become a fundamental aspect of interacting with modern software systems.

Despite its importance, reverse engineering remains largely an *art*. Analysts develop intuition through experience, the community evaluates tools using inconsistent or task-specific metrics, and researchers often judge success indirectly through downstream outcomes such as vulnerability discovery or patching. This lack of formalization limits our ability to reason about progress, compare techniques, and design systems that reliably operate on unfamiliar software.

Reverse engineering is fundamentally a process of *software understanding*. Given an artifact and incomplete context, the analyst reconstructs representations that explain behavior and support meaningful interaction. This notion of understanding remains implicit and inconsistently defined across prior work, leading to fragmentation across tools, methodolo-

gies, and evaluation strategies.

1.0.1 Toward a Science of Software Reverse Engineering

In this dissertation, I argue that reverse engineering can be transformed into a science by systematically studying its techniques, processes, and applications, grounded in a principled notion of software understanding. I define software understanding as:

> **The ability to reconstruct, explain, and modify the behavior of a software system without access to its original design.**

This definition captures the central objective of reverse engineering. Understanding is *reconstructive*: recovering representations that approximate original intent. Understanding is *explanatory*: enabling reasoning about behavior. Finally, understanding is *actionable*: supporting modification and intervention.

Establishing a science of reverse engineering, therefore, requires systematically studying three interconnected aspects of the discipline. First, we must study the individual *techniques* that enable reverse engineering, defining principled metrics for their quality and designing approaches that achieve those metrics. Second, we must study the *processes* by which analysts compose multiple techniques into effective workflows, particularly as these processes increasingly integrate automated systems such as LLMs. Third, we must study the *applications* of reverse engineering, since these applications ultimately define what aspects of the field we should optimize and whether our metrics capture real-world utility.

1.0.2 Problem Statement

Reverse engineering often operates as the inverse of software construction processes, which are inherently lossy or opaque. Compilation, abstraction, optimization, and automated code generation transform software in ways that obscure original structure and

intent. Even when source code is available, important context, such as design rationale, invariants, or assumptions, is often missing.

This setting introduces several core challenges. First, the individual techniques of reverse engineering lack principled metrics for success. Existing tools often optimize for surrogate objectives, such as readability or structural simplicity, which do not necessarily correspond to fidelity to the original program. For example, decompiler transformations that simplify control flow may improve readability while deviating from the structure intended by the original developer.

Second, the reverse engineering process, specifically how analysts compose multiple techniques into effective workflows, remains poorly characterized. Analysts construct understanding through iterative exploration, hypothesis formation, and validation, combining diverse techniques at each step. With the emergence of LLM-assisted workflows, this process increasingly involves collaboration between humans and automated systems, yet is difficult to study and measure.

Third, the applications of reverse engineering are rarely examined as a driver of how techniques should be designed and evaluated. Tasks such as vulnerability discovery, patch generation, and system modification ultimately define the utility of reverse engineering, yet the connection between application demands and technique design remains underexplored.

Taken together, these challenges highlight the absence of a unified framework for reasoning about reverse engineering across its techniques, processes, and applications.

1.0.3 Overview of This Dissertation

This dissertation develops a principled foundation for reverse engineering by studying its techniques, processes, and applications.

Studying Reverse Engineering Techniques

I first study reverse engineering techniques by addressing the lack of principled metrics for evaluating their quality. Existing approaches, particularly in decompilation, often rely on ad hoc measures that do not capture fidelity to the original program.

To address this, I introduce *closeness-to-source* as a goal of decompilation, and Graph Edit Distance (GED) as a quantitative metric for measuring decompilation quality. I show that principled metrics not only enable rigorous evaluation but also reveal how techniques must be redesigned to achieve their goals.

Through a detailed analysis of decompilation failures, I identify compiler transformations as a fundamental source of structural distortion. I then design and implement **SAILR**, a compiler-aware decompilation framework that explicitly inverts these transformations. To support this, I develop a full decompiler within the angr framework, enabling controlled evaluation and reproducibility.

This contribution demonstrates that advancing reverse engineering requires both defining rigorous metrics for individual techniques and designing systems that achieve them.

Studying the Reverse Engineering Process

I next study the reverse engineering process, examining how analysts compose multiple techniques into effective workflows. I conduct an empirical study of reverse engineering workflows, observing how humans iteratively apply and combine techniques both independently and with LLM assistance. I introduce methodologies for evaluating process effectiveness through task performance, capturing correctness, efficiency, and interaction strategies.

My findings show that the reverse engineering process is dynamic and strategy-dependent. LLMs significantly alter how analysts compose techniques, improving performance in cer-

tain settings but also introducing new failure modes. These results highlight the importance of studying reverse engineering not only through individual techniques but through the processes by which they are combined.

Studying Reverse Engineering Applications

Finally, I study the applications of reverse engineering, which ultimately define the utility of our techniques and metrics. I present **PatcherY**, an autonomous patching system that applies automated reverse engineering to identify vulnerabilities, generate candidate fixes, and validate their correctness.

PatcherY demonstrates that application demands provide a concrete test of whether reverse engineering techniques and metrics serve real-world needs. By operating in settings where systems must be modified without full prior understanding, PatcherY reveals which aspects of understanding are essential for successful intervention and which are not.

This contribution establishes that studying applications is essential to a science of reverse engineering, as applications define what our techniques should optimize for and whether our metrics capture what matters.

1.0.4 Thesis

The central thesis of this dissertation is:

> **Software reverse engineering can be transformed into a science by studying its techniques through principled metrics, characterizing the processes by which those techniques compose into practical workflows, and examining the applications that define reverse engineering's utility and validate its methods.**

1.0.5 Contributions

This dissertation makes the following contributions:

- SAILR, a compiler-aware decompilation framework and full decompiler implementation, demonstrating how principled metrics can drive the design and evaluation of reverse engineering techniques.
- An empirical study of human and LLM-assisted reverse engineering, providing new insight into the processes by which analysts compose techniques into effective workflows.
- PatcherY, an autonomous patching system demonstrating how applications of reverse engineering validate and inform the design of techniques and metrics.
- A unified perspective connecting techniques, processes, and applications into a coherent foundation for software reverse engineering.

Chapter 2

STUDYING REVERSE ENGINEERING TECHNIQUES

2.1 Introduction

Compiled programs called binaries power the devices that drive modern society. However, many of these binaries are shipped without accompanying source code, making their analysis arduous. To reduce this burden, decompilation techniques aim to recover source code from binary code. Security analysts use decompilation in reverse engineering [19, 70], malware analysis [33, 32, 106, 77], and vulnerability discovery and mitigation [72, 90].

Many applications of decompilation require *high-quality* decompiled source code. One important criterion of high-quality source code is meaningful control flow structures, such as `if-else` statements, `while` loops, and `do-while` loops in the C language. Unfortunately, the semantics of these control flow structures are lost during compilation, replaced by simple binary-level control flow transfers such as `jmp` instructions.

Decompilers leverage control flow *structuring* algorithms that analyze low-level constructs in the compiled binary and attempt to recover the high-level control flow. If compilers simply translated C code to assembly and pieced the assembly together with `jmp` instructions, the resulting code would be easily *structurable*, and decompilers would be able to produce high-quality decompiled source code. However, modern compilers optimize and distort code structures during compilation, making the result *unstructurable* and preventing current structuring algorithms from recovering the high-level control flow structure.

Structuring failures manifest as *goto* statements in the decompilation, stitching together portions of code where the decompiler failed to recover reasonable high-level struc-

tures [107]. The fact that it is specifically a goto that results from such failures has caused some semantic confusion between the Software Engineering and Decompilation research communities. In Software Engineering, *gotos are considered harmful*, with Dijkstra famously stating that "the quality of programmers is a decreasing function of the density of goto statements in the programs they produce" [31]. Treating decompilers as *software engineers*, and thus treating gotos as indicators of low-quality output, recent research into structuring algorithms has focused on extensively restructuring decompiled code to reduce or eliminate goto statements [17, 107, 50].

However, gotos *do* exist in popular C codebases, such as the Linux kernel, which, as of version 6.1, contains 3,754 gotos. Intuitively, previous work that removes all gotos will decrease the quality of decompiled code when it removes *intended, developer-written* gotos that are in the original source code, as their elimination induces differences between the source code and the decompilation. Clearly, such gotos, *in the context of faithful decompilation*, are not harmful.

In fact, the lack of decompiler consideration for intended gotos is a symptom of a more general weakness in decompilation approaches. Existing techniques tackle the structure of binary code without a principled understanding of how this structure came to be. Our intuition is that an understanding of the differences between intended and spurious gotos in decompiled code will shed light on the binary constructs that hamper code structuring during decompilation.

This work is the first to investigate and identify the actual causes of spurious gotos. We systematically studied why spurious gotos appear in decompiled code, tracking them down to wide arrays of optimizations leveraged by the compiler. We investigated all compiler optimizations at level `O2` and below in GCC 9.5 [1] on Coreutils and identified all optimizations that may introduce unstructurable code. To our surprise, some of these optimizations

[1] We also analyzed LLVM and identified analogous optimizations.

(responsible for around 17% of spurious gotos) are even enabled in *non-optimized* (`O0`) mode and cannot be easily disabled.

This investigation led us to a realization: To preserve the intended structure, including source-present gotos, and eliminate sources of gotos introduced by the compilation process, decompilers can leverage structuring algorithms informed by compiler-derived knowledge. Based on our findings, we created the first compiler-aware control flow structuring algorithm, SAILR [2] , that uses compiler-derived knowledge to *deoptimize* binary code and make unstructurable code structurable. SAILR mirrors the compilation pipeline of GCC and precisely inverts goto-introducing transformations rather than using broad (and imprecise) goto-eliminating modifications leveraged by prior work.

SAILR generates decompilation that, compared to state-of-the-art techniques, contains significantly fewer spurious gotos while maintaining high structure similarity to the source code. For example, among other evaluations, we evaluated decompilers against 7,355 functions (containing 1,367 intended gotos) from 26 popular C Debian packages. The state-of-the-art Hex-Rays decompiler included in IDA Pro [3] emitted 6,115 gotos while achieving an average Graph Edit Distance (GED) of 22.52 between the decompilation and the source code. SAILR reduced the gotos count to 2,673 (we discuss the remaining gotos in Section 2.9) while maintaining an average GED of 22.64. DREAM, a state-of-the-art structuring technique that guarantees the elimination of *all* (including intended) gotos, emitted 0 gotos and significantly impacted the quality of the decompilation, achieving an average GED of 45.99. An optimal decompiler should have both the same number of gotos as the source and a GED score of zero.

Furthermore, while comparing SAILR against existing research, we realized that existing structuring techniques are either not open-sourced or have bit-rotted, hampering research in the field. To enable impartial comparative studies, we built a new decompiler for

[2]SAILR: Structured AIL Reverter.

`angr`, which we refer to as the ANGR DECOMPILER hereinafter, and implemented SAILR as well as other previous structuring algorithms (namely, Phoenix [17], DREAM [107], and rev.ng's Combing [50]) in it. Due to its Python implementation, our decompiler can be significantly slower than other decompilers, however, we expect performance to improve as it is further developed. We hope the ANGR DECOMPILER will foster future decompilation research, improve the reproducibility of decompilation techniques, and make future work in this area easier to approach.

Contributions. This work makes the following contributions:

- We identify the root causes of unstructurable code during the decompilation of C: compiler optimizations and knowledge gaps between the decompiler and the compiler.
- We design SAILR, a novel compiler-aware structuring algorithm that precisely inverts the root causes of goto-introducing compiler transformations to generate decompilation significantly closer to the original code.
- We build an open-source decompiler ANGR DECOMPILER, and then implement SAILR as well as three other structuring algorithms on our decompiler for an impartial evaluation.
- We design a new set of evaluation metrics to measure the quality of control flow structuring during decompilation by measuring how close it is to high-quality source code. Using our metrics, we evaluate SAILR and demonstrate its improved structuring performance over other structuring algorithms and state-of-the-art decompilers.

In the spirit of open science, we are *actively* developing the ANGR DECOMPILER and SAILR as open-source projects [3] . We have also open-sourced our evaluation framework, which includes the specific versions of `angr` and SAILR used in our evaluation [4] .

[3] https://github.com/angr/angr

[4] https://github.com/mahaloz/sailr-eval

2.2 Background & Motivation

Before diving into the technical details of SAILR, we introduce necessary background knowledge about compiling C programs (Section 2.2.1), decompiling C programs (Section 2.2.2), state-of-the-art control flow structuring algorithms (Section 2.2.3), and the motivation of our research (Section 2.2.4).

2.2.1 Modern C Compilation

Modern C compilation has two high-level phases relevant to the understanding of the decompilation process. First, the source code is "lifted" into an Intermediate Language (IL) as C is too high level to be operated on directly. In the case of GCC, C code is "lifted" into an IL called Gimple [2].

Once lifted, optimizations are applied to the IL. Optimizations that are specific to a program's control flow use *schemas* to identify optimization locations. Schemas are similar to puzzle pieces for a program's control flow. Once a schema is found to fit a certain control flow archetype, such as a while loop, the corresponding operation can be performed on the IL to manipulate it into an optimized form.

2.2.2 Modern Binary Decompilation

Modern binary decompilers generally perform three phases of analysis before generating C-style code.

Control flow graph recovery. A decompiler first disassembles machine code in a binary to recover assembly instructions, function boundaries, and execution flows. Control flow information is stored in a control flow graph (CFG), with basic blocks as vertices and execution flow as edges. Optionally, the decompiler may lift instructions into an IL to assist with architecture- and platform-agnostic analysis.

```
long long schedule_job(unsigned int
    a0, unsigned int a1, unsigned
    int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT != a2)
        {
            next_job();
            refresh_jobs();
        }
    }

    if (!a0 || !a1)
        refresh_jobs();
    if (a1 && (!a0 || EARLY_EXIT !=
        a2))
        fast_unlock();

    complete_job();
    log_workers();
    return job_status(a1);
}
```

```
long long schedule_job(unsigned int a0,
    unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
        refresh_jobs();
        goto LABEL_4012d3;
    }
    refresh_jobs();
    if (!a1)
        goto LABEL_4012eb;
LABEL_4012d3:
    fast_unlock();
LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

```
long long schedule_job(unsigned int
    a0, unsigned int a1, unsigned
    int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
    }
    refresh_jobs();


    if (a1)
        fast_unlock();


LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

Figure 2.1: (From left to right) the DREAM, Phoenix, and SAILR decompilation of Listing 1 (using GCC 9.5 -O2).

Type inference. After recovering a CFG, the decompiler uses parts of the graph to infer variable locations and their associated types. Prior research has explored various methodologies for inferring types, such as TIE [66], Howard [96], and retypd [84]. More recently, Osprey [114] and DIRTY [20] utilize machine learning techniques to predict types. Optionally, this inference phase may also recover prototypes and calling conventions for all functions.

Control flow structuring. In this last phase, the decompiler will transform assembly instructions or IL statements into C-style statements using patterns and flow information from the CFG. This phase will also simplify these statements and identify common compiler idioms to replace them with more human-friendly representations. In general, a CFG can be structured in many different, but semantically equivalent, forms, making choosing the right structure a difficult task.

Table 2.1: A taxonomy of control flow structuring algorithms in modern decompilers and decompilation literature. "Duplication Removal" refers to removing compiler-introduced code duplication. An optimal compiler will remove duplicates and have no redundant code/conditions. We observed that in rare cases Hex-Rays will remove exactly duplicate blocks.

Decompiler	Condition Aware	Emit Gotos	Duplication Removal	Open-source	Redundant Conditions	Redundant Code
Hex-Rays [3]	Yes	Yes	Rare	No	No	No
Ghidra [79]	Yes	Yes	No	Yes	No	No
mirtoc (extended) [36]	No	Yes	No	No	No	No
Phoenix [17]	Yes	Yes	No	No	No	No
DREAM [107]	Yes	No	No	Yes	Yes	No
rev.ng [50]	Yes	No	No	No	Yes	Yes
SAILR	Yes	Yes	Yes	Yes	No	No

2.2.3 Structuring in C Decompilation

Table 2.1 shows a taxonomy of existing control-flow structuring algorithms. We broadly categorize structuring algorithms into two groups:

Graph-schema-matching algorithms. These structuring algorithms attempt to match subgraphs of a CFG against known control-flow patterns for C control-flow structures. For example, a diamond-shaped subgraph commonly corresponds to an `if-else` construct when the two middle nodes have inverted edge conditions. Being condition-aware is critical to the correctness of these structuring algorithms [17].

A key challenge that these algorithms face is unknown graph schemas. Structuring algorithm authors usually hardcode graph schemas for common C control-flow constructs. However, because compiler optimizations create novel graph schemas, there are more graph

```
1  int schedule_job(int needs_next, int fast_job, int mode)
2  {
3      if (needs_next && fast_job) {
4          complete_job();
5          if (mode == EARLY_EXIT)
6              goto cleanup;
7
8          next_job();
9      }
10
11     refresh_jobs();
12     if (fast_job)
13         fast_unlock();
14
15 cleanup:
16     complete_job();
17     log_workers();
18     return job_status(fast_job);
19 }
```

Listing 1: A motivating example based on code from the Linux kernel job scheduler.

schemas in real-world binaries than one can reasonably enumerate. As a result, these algorithms must virtualize edges (i.e., eliminating edges and replacing them with spurious gotos) to create more subgraphs that would match against known graph schemas, which significantly hampers the quality of decompiled code.

Goto-less algorithms. These structuring algorithms (used by DREAM and rev.ng) do not use edge virtualization to convert non-schema-matching subgraphs into schema-matching ones. Instead, they create new control-flow structures that are functionally equivalent to non-schema-matching subgraphs. These new structures often do not match the source's because they can introduce new or duplicated code. DREAM duplicates code found in conditions while moving around contained logic. rev.ng duplicates code found in incomplete structures to make them match known schemas. These algorithms rely heavily on

block condition information to make goto-less structures.

2.2.4 Motivation

At first glance, goto-less algorithms may appear superior to graph-schema-matching algorithms since they never generate spurious gotos. However, between the two structuring algorithms, goto-less approaches often differ more from the source's control flow structure. We assume that C source code in a popular and actively maintained codebase (such as code in GNU packages) is a high-quality implementation of program logic. Therefore, when decompiling binaries that are compiled from high-quality code, decompilation that is structurally close to the source is of high quality. Using source code structure as the ground truth, we manually evaluated modern structuring approaches and found their results to be significantly flawed.

Listing 1 shows a motivating example based on code found in the Linux kernel scheduler. Figure 2.1 shows the DREAM, Phoenix, and SAILR decompilation, respectively, for the compiled binary from Listing 1. While there are no spurious gotos in the DREAM decompilation, the `if` statements contain complex and irreducible conditions that make understanding execution flows difficult. DREAM decompilation contains duplicated conditions that did not exist in the source on Lines 13 and 15. Although this duplication decreases code complexity compared to Phoenix, it obfuscates the original programmer's intentions. It also makes the needed conditions at each line of code, like Line 16, harder to understand.

In the Phoenix decompilation, although it contains a goto found in the source code, it also contains two spurious gotos. The gotos targeting `LABEL_4012eb` are due to a lack of special-case `if`-statement schema in Phoenix. The goto targeting `LABEL_4012d3` is due to a lack of compiler-optimization knowledge. The compiler introduced extra code and created a path to skip an extra conditional instruction using the Jump Threading optimiza-

tion (see Section 2.3.3). This optimization introduces an extra call to `refresh_jobs` in both the Phoenix and DREAM decompilation.

All the decompilers that we tested failed to remove the duplicated statements in this example. Additionally, any schema-based structuring algorithms, like those used in Ghidra and Hex-Rays, cannot remove the goto targeting `LABEL_4012d3`. This is because no schema exists to structure this goto edge. To generate high-quality decompiled code, we must develop a new structuring algorithm that can undo the compiler's transformations, remove the duplicated blocks, and generate high-quality decompilation that is close to the original source. The last code snippet in Figure 2.1 shows the decompilation result of such a new structuring algorithm.

2.3 Goto-Inducing Compiler Transformations

Without considering goto statements in source code, each C construct can be represented by a single-entry, single-exit (SESE) graph whose nodes are basic blocks in the construct [36]. Compilers rely on a finite number of graph schemas during the code generation phase (Section 2.2.1). If we split the binary CFG of a function into nested SESE graph regions, a graph-schema-matching structuring algorithm matches each graph region against a known schema as long as it knows *all* graph schemas that a compiler uses. This works for binaries built without *any* compiler optimizations.

Unfortunately, some compiler optimizations split, merge, or duplicate nodes on graphs without preserving graph schemas. These optimizations create new graph schemas and make it impossible for a structuring algorithm to exhaustively enumerate all possible graph schemas that an optimization compiler may generate. We find that there are three reasons for edge virtualization to happen during structuring:

- (Real) gotos that exist in the source code. An optimal structuring algorithm will want to preserve these gotos.

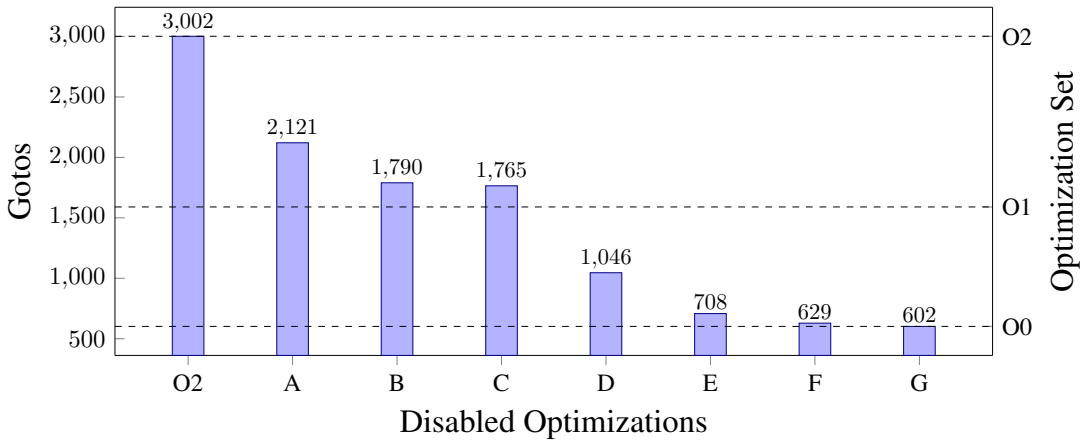

Figure 2.2: Gotos present in Hex-Rays decompilation as optimizations in Section 2.3.3 are disabled. Each optimization point disables itself and all optimizations to its left. Optimization sets `O2` through `O0` are shown for reference.

- Graph schemas that compilers use during code generation (without optimizations) but are not known to structuring algorithms. Because there are a finite number of these schemas, an optimal structuring algorithm should know all of them.
- New graph schemas that compiler optimizations create. An optimal structuring algorithm should remove all resulting gotos by reverting these optimizations.

We pick Coreutils 9.1 as the high-quality C codebase and GCC 9.5 as the compiler for our studies.

2.3.1 Enumerating Missing Graph Schemas

While one could read all of the source code of a compiler to find all unsupported graph schemas in a decompiler, this approach does not scale. We propose the following methodology to identify schemas missing in a decompiler.

First, we collect a set of binaries from high-quality codebases and compile them using GCC with optimizations (`O2`) while enabling the `save-temps` and `dump-tree-all` flags. These flags ask the compiler to save *intermediate files* that are generated during preprocessing and optimization passes. Next, we decompile all functions in all binaries and

identify functions with gotos in the decompilation but not the source. For each function, we manually note the locations of the gotos in the function and perform a binary search on the intermediate files to identify which optimization pass changed the function to induce the goto. This identifies the GCC source for the optimization pass, and these passes are well-commented with information on the schema used.

2.3.2 Measuring Goto Introduction

Once we identify an optimization pass that causes an unstructurable subgraph, we attempt to understand the impact scale (i.e., how many gotos this optimization induces for binaries in Coreutils). We disable the optimization and repeat the steps in Section 2.3.1 until the goto disappears from the final decompilation. If the goto disappears, we count all the gotos in the decompilation before and after the optimization to measure its impact. We group all the disabled optimizations that are required to make the goto disappear as either one optimization algorithm or a set of tightly coupled algorithms.

This method worked surprisingly well for most optimizations in the GCC `O2` optimization set. It is worth mentioning that some optimizations did not disappear after disabling their frontend optimization option; These optimizations could only be entirely disabled by undocumented developer options.

2.3.3 Goto-Inducing Transformations in GCC

In an analysis of 1,150 unique functions across 135 object files in Coreutils 9.1, we found the cause of every goto across the 3,002 gotos that are present in the decompilation of Hex-Rays 8.0. We compiled these object files with GCC 9.5 with optimization level `O2`, inlining disabled (for function tracking), debug symbols enabled, and intermediate file dumping. We filtered these functions for uniqueness by eliminating functions that shared the same name, with the exception of common symbols such as `main` and `usage`.

We associate every goto to a specific GCC transformation using the methodology in Section 2.3.2. These transformations include togglable optimizations, internal highly-coupled optimizations, and compiler knowledge exploitation.

Jump Threading (A). When one branch's conditions superset a future branch's conditions, the statements contained between the two may be duplicated into the first and end with a jump to avoid extra conditional instructions. Gotos appear between duplicated statements.

Common Subexpression Elimination (B). The compiler finds common statements among multiple blocks and reduces them into one use of the expression and a series of jumps to that expression. Gotos occur between condensed statements.

Switch Conversion (C). The compiler replaces simple assignments on switch statements in cases with assignments from scalars. Gotos can occur between cases that share a common expression for assignment.

Cross Jumping (D). Unifies equivalent code (e.g., repeated statements) across regions and replaces duplicates with a jump to the unification. A goto corresponds to the new jump.

Software Thread Cache Reordering (E). The compiler estimates the likelihood of executing a set of paths and clusters those frequently executed together through code duplication.

Loop Header Optimizations (F). The compiler moves branches that are always true or true only once to avoid executing a conditional instruction many times. The edge leaving the loop body (for the copied header) becomes a goto.

Builtin Inlining (G). The compiler replaces special built-in functions, e.g., `strcmp`, with optimized inlining and propagation. Gotos can occur when inlining happens inside a short-circuit Boolean expression.

Switch Lowering (H). A highly-coupled optimization that is non-disablable in GCC. The compiler optimizes switch statements by breaking them into clusters and applying heuristics to avoid large jump tables. Gotos occur when the switch statement is fully transformed into a nested `if` statement.

Nonreturning Functions (I). Some functions, such as `exit` and `abort`, may not (always) return, and GCC uses this knowledge to transform the CFG. When the decompiler lacks knowledge of these functions' ability to not return, it can cause successors to be incorrectly added during CFG recovery, causing goto edges to those successors.

Figure 2.2 shows the number of gotos present in decompilation after disabling transformations in compounding order. In addition to associating the cause of every goto, we group the goto-inducing optimizations into two classes by effect. Both classes broadly transform *irreducible statements*, which we define as statements found in the source code that cannot be eliminated. Such statements must always exist for the *same path* they were found in the source, though the literal instructions that cause them may be condensed.

Irreducible Statement Duplication (ISD) converts a statement into many statements that are semantically equivalent to the original. This set includes optimizations A, E, and F.

Irreducible Statement Condensing (ISC) converts many statements into a condensed version with introduced graph edges. This set includes optimizations B, C, and D.

Optimizations G, H, and I do not fit well into either ISD or ISC optimizations and instead are classified as miscellaneous optimizations. These miscellaneous optimizations account for a minority of goto-inducing optimizations and require more specific algorithms to revert.

2.4 Overview of SAILR

We implemented our structuring algorithm, SAILR, and our decompiler, ANGR DECOMPILER, on top of the `angr` binary analysis platform [95]. As such, our decompiler supports any architecture `angr` supports. The ANGR DECOMPILER handles miscellaneous decompiler tasks such as program lifting while SAILR structures lifted Control Flow Graphs (CFG).

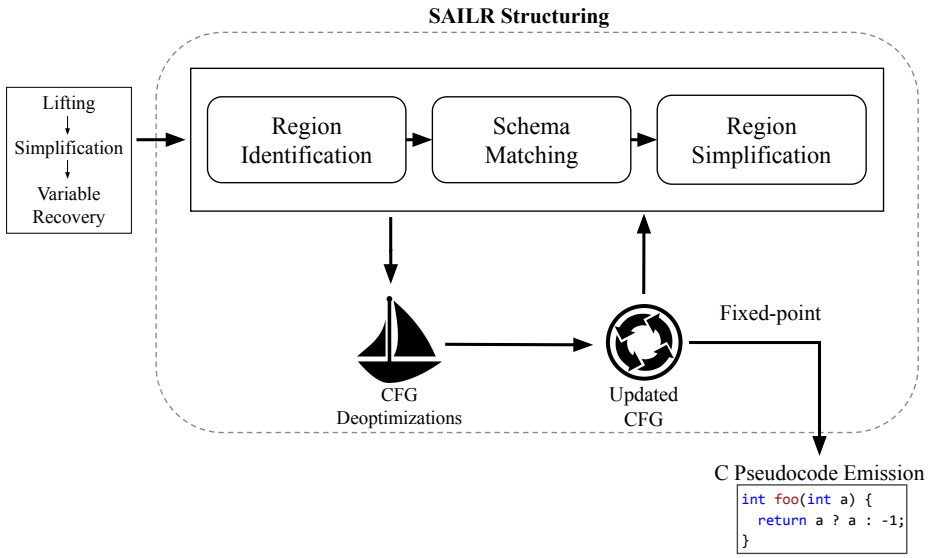

Figure 2.3: Overview of ANGR DECOMPILER's decompilation pipeline.

2.4.1 The Decompilation Pipeline

The ANGR DECOMPILER is similar in design to prior work [17, 107]. Using `angr`, we convert a binary into a lifted CFG. The lifted CFG contains VEX Intermediate Representation (IR) instructions, which is the IR `angr` uses. Figure 2.3 illustrates the high-level steps in the ANGR DECOMPILER after a VEX IR CFG is provided.

Lifting, Simplification, and Variable Recovery

The ANGR DECOMPILER takes a function-level VEX IR CFG as input. Because VEX IR cannot represent C-style expressions, we designed a more abstractable intermediate language called the ANGR INTERMEDIATE LANGUAGE (AIL). The ANGR DECOMPILER starts by lifting the VEX IR CFG into an AIL CFG. Multiple rounds of simplifications are run on the AIL CFG to eliminate redundancy. This lifting process is similar to GCC's simplification process while lifting C to Gimple. After lifting, the ANGR DECOMPILER recovers variable locations, function prototypes, and variable types using the AIL CFG.

SAILR Control Flow Structuring

After collecting an AIL CFG and variable information, the ANGR DECOMPILER uses the SAILR structuring algorithm to turn the AIL CFG into C pseudocode. The following four passes are run until a fixed point is reached. These passes together make up the core of the SAILR structuring algorithm.

Region identification. The region identification algorithm in the ANGR DECOMPILER is based on the algorithm that DREAM employs [107]. First, it identifies all single-entry, single-exit (SESE) regions in an AIL graph, following a reverse topological order of all nodes. As its name suggests, a SESE region refers to a group of AIL blocks and edges that contain one entry and one exit. A region may contain multiple regions.

Schema matching. The ANGR DECOMPILER then preliminarily structures each region using a graph-schema-matching algorithm built using concepts developed in Phoenix [17]. A major difference between SAILR and Phoenix is that Phoenix directly matches schemas against subgraphs on the full graph while SAILR matches schemas on a per-region basis. SAILR matches subgraphs in each region against known graph schemas.

When SAILR fails to match a region against any known graph schemas, it will virtualize an edge by removing it from the region and replacing the source jump statement with a goto. Then, SAILR condenses blocks that match a known graph schema into a C-style construct. Once the ANGR DECOMPILER finishes structuring a region, it replaces the region in its parent region with the structured node before continuing to structure its parent region. Structuring terminates once the outermost region is structured.

Region simplification. Next, the ANGR DECOMPILER performs simplification and optimization on the AIL graph of each refined region to eliminate redundant edges and blocks that may not affect the control flow. This is also applied to the preliminarily schema-matched result.

Deoptimization. In the last step of SAILR structuring, the ANGR DECOMPILER performs deoptimizations directly on the AIL graph. The deoptimization phase relies on all knowledge previously collected in the pipeline, such as region information, goto locations, and reaching conditions. Each deoptimization described in Section 2.5 is run iteratively or until a limit is met.

C Pseudocode Emission

Finally, the ANGR DECOMPILER expands the structured regions into a C-style abstract syntax tree (AST) and pretty-prints it as a function with linear C-style statements. One novel aspect in the decompilation pipeline is that we stall optimizations on the lifted function CFG until the last point of the process. This allows CFG edits, which are the basis for SAILR, to have the most available knowledge.

2.4.2 Major Contributions of SAILR

Previous works in control flow structuring have focused on making decompilation more structured [17, 107, 50]. The Phoenix [17] decompiler identified this unstructurability through gotos. Phoenix focused on reducing these gotos through condition-aware graph schema matching. DREAM [107] eliminated gotos by duplicating conditions across the code to create bounded statements. rev.ng [50] also eliminated gotos through duplication, but instead duplicated executable code, not conditions on nodes. In the case of DREAM and rev.ng, these approaches generically eliminate all gotos regardless of their existence in the source.

In contrast, SAILR does not blindly eliminate gotos and instead treats the cause: compiler transformations. SAILR precisely reverts compiler transformations found to be the cause of unstructurable code, which manifest as gotos in decompilation. Of these transformations, certain compiler optimizations and the gap between the decompiler and the

compiler play a significant role in unstructurability. SAILR approaches a solution to both of these problems by improving the knowledge of the decompiler and reverting certain optimizations.

Reverting compiler optimizations. SAILR's deoptimization passes revert the most impactful optimizations to control flow structure. At a high level, SAILR directly edits the AIL graph by condensing (Section 2.5.1), duplicating (Section 2.5.2), or moving (Section 2.5.3) nodes. In each case, SAILR uses information derived from GCC 9.5 to perform the reverse actions of these optimizations. For each deoptimization pass described in Section 2.5, the majority of computation time comes from searching for valid cases. This search phase is *goto-guided*. We assume all optimizations that SAILR wants to deoptimize are goto-inducing, so we only search the blocks around a goto edge when finding candidates. Although the understanding of these optimization techniques is based on GCC, we find that other compilers, such as Clang, implement the same optimizations in similar ways.

Closing the compiler-to-decompiler knowledge gap. We identified two critical knowledge gaps by studying the Hex-Rays decompilation of Coreutils `O2` binaries compiled by GCC 9.5. The first gap is the *returning* of a callee, i.e., whether a callee function returns or not. A compiler will not generate any return site for a call if it knows the callee never returns, for example, *_exit()*. The function CFG that any decompiler recovers will be different from the compiler's CFG if they are not aware of this fact, which may lead to unstructurable code. This problem is further complicated because the returning can be call-site specific: A callee function may be returning in one call site and non-returning in a different site, which we will detail in Section 2.6.1. The second gap is the decompiler's unawareness of all the graph schema the compiler uses. We discuss major ones in Section 2.6.2.

2.5 Deoptimizing Decompilation

In this section, we give an overview of novel algorithms to deoptimize decompilation. These algorithms revert three types of optimizations discussed in Section 2.3.3: ISD, ISC, and miscellaneous optimizations. In each deoptimization, there is a *search* and *transformation* phase that parallel the compiler's optimization process. The search phase is generally more computationally intensive.

2.5.1 ISD Optimizations

The main effect of ISD optimizations is the duplication of subgraphs found in the original source CFG. Most compilers will limit the number of statements that can be duplicated when performing optimizations in this class. During GCC's ISD optimization passes, reaching conditions are computed for each statement in the CFG. Reaching conditions determine what statements are accessible when a condition holds during execution. For each optimization, GCC further determines if a statement is duplicatable using other indicators. In the case of Jump Threading, the indicator is whether an ancestor statement shares an overlapping condition that guarantees the execution of the current statement. For optimizations such as Software Thread Cache Reordering, the duplication indicator can be as complicated as computing if a block has a higher execution probability over its neighbors.

Next, the compiler will duplicate both the marked statement and the graph found under that reaching condition. The duplication will be placed at another location in the graph that guarantees the same reaching conditions, guarded by either already present blocks or new conditional blocks. The duplicated subgraph may have other statements appended to the graph tail nodes. Finally, the duplicated subgraph tail nodes are connected to the original subgraph successors by an edge.

Figure 2.4 shows a C program that when compiled with GCC `O2` optimizations results

```
void foo(int a, int b) {
  if (a && b) {
    puts("first print");
  }



  puts("second print");
  if (b) {
    puts("third print");
  }


  sleep(1);
  puts("leaving foo...");
}
```

```
void foo(int a, int b) {
  if (a && b) {
    puts("first print");
    puts("second print");
    goto label_1;
  }
  puts("second print");
  if (b) {
label_1:
    puts("third print");
  }
  sleep(1);
  puts("leaving foo...");
}
```

Figure 2.4: Example C code shown before and after transformation from Jump Threading, an ISD optimization. The second condition of the original code is always true if the first condition is true, causing the comparison to be subverted by a jump.

in optimized output. The compiler computes the conditions for both the first `print` and the third `print`. Because the conditions overlap, the compiler determines that both scopes will execute if the first condition (a && b) holds true. The compiler then duplicates the code between the two conditions and creates a jump (a goto) from the first condition scope to the second condition scope. We next discuss how SAILR condenses ISD-duplicated blocks.

Finding duplicate statements. Given an AIL function CFG, SAILR starts by iterating through all combinations of two statements and creates an initial set of candidate statement pairs that exist in the same region and are *storage equivalent*. Two statements are storage equivalent when all their reads and writes to locations are the same. For a pair of call statements, storage equivalence means all arguments share the same storage locations and the function addresses are the same.

With the candidate set, we take all candidates and expand their similarity match by expanding the connected graph of the matching statements. We assume the first found

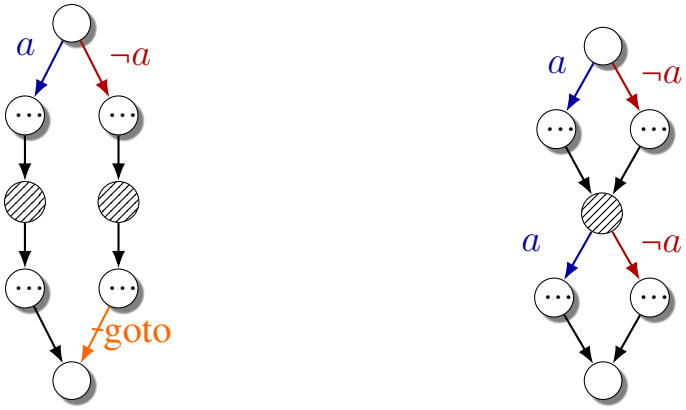

Figure 2.5: CFGs before and after deoptimizing an ISD optimization case. In order to identify an ISD case, the shaded node must be found to have a semantic duplicate, as well as post-dominating goto edge. The nodes are merged and then bounded by their previous conditions.

statement of the pair is the head of each similar subgraph. Then we iteratively expand each subgraph using a Breadth-First Search (BFS) traversal and match the statements using the Knuth-Morris-Pratt (KMP) algorithm [63].

Finally, with each candidate expanded to its maximum graph similarity, we find the tail nodes of each duplicate graph. All of these nodes must share the same successor in each graph. At least one of the two graphs must have a goto edge, which was identified during structuring, that connects it to the common successor. In Figure 2.5 (left) the graph of a matching case is shown. Note, that any number of nodes can exist between the condition head and the duplicated node.

Condensing duplicates. The key to condensing a pair of candidate subgraphs while preserving the original semantics is maintaining the reaching conditions of every node in the graph after condensing. In Figure 2.5 (right), a graph shows the correct result of reverting an ISD case.

First, the duplicated node must be merged into one node by redirecting all predecessors to the newly merged node. Although shown as a single node in this example, the merged node can be a graph, which means the predecessors may go to multiple nodes. Next, the

Figure 2.6: CFGs before and after deoptimizing an ISC optimization case. ISC cases contain a goto edge connecting one node to another node that has multiple predecessors. Duplication of the shaded node, and all its single-successor ancestors, revert this case.

condition a must be used to correct the flow of execution before and after leaving the merged node. If there are multiple head nodes in the merged sub-graph, each head node must have the correct a condition blocking its path. In the case of a single node being the head of a merged graph, only the exits of the merged sub-graph need duplicated conditions. Finally, the graph is in the deoptimized state and should no longer have a goto. Depending on the conditions before and after the ISD case, we can simplify the exiting conditions on the merged sub-graph. Listing 3 in the Appendix shows the pseudocode of our ISD deoptimization algorithm.

2.5.2 ISC Optimizations

The main effect of ISC optimizations is the reduction of subgraphs found in the original source CFG through condensing. Condensing turns n equivalent statements into $n - m$ statements, where $m \geq 1$. Similar to ISD optimizations, ISC optimizations require knowledge of reaching conditions and statement semantics. ISC optimizations traverse the CFG for semantically equivalent pairs of statements (or subgraphs). Then, for each statement pair, ISC optimizations will eliminate one statement and connect the other with a jump to the remaining statement.

Figure 2.7 shows an example C program that exhibits gotos when optimized with Cross

```
int foo(int a, int b) {
  if(!a)
    return -1;
  puts("first print");
  if(!b) {
    return -1;
  }


  puts("leaving foo...");


  return 1;
}
```

```
int foo(int a, int b) {
  if(!a)
    goto label_1;
  puts("first print");
  if(!b) {
label_1:
    ret = -1;
    goto label_2;
  }
  puts("leaving foo...");
  ret = 1;
label_2:
  return ret;
}
```

Figure 2.7: Example C code shown before and after transformation from Cross Jumping, an ISC optimization. In the original code, the *return* statement, as well as its return value, are reused in the same execution pass resulting in statements being merged and connected with a goto.

Jumping, an ISC optimization. The `return` statement is reused across many if-statements, which is a common programming pattern. In the resulting code output, all but one `return` is replaced with a goto. For the first `if`, Cross Jumping also eliminates the constant assignment to `ret`. We next discuss how SAILR duplicates ISC-condensed blocks to recover the original structure.

Finding condensed statements. Finding condensed statements requires finding the locations of goto edges (after structuring) in an AIL CFG. Similar to searching for ISD cases, this demands that a decompiler can iteratively perform graph optimizations and structuring. We can only know the locations of gotos in an AIL function CFG after structuring at least once. Figure 2.6 (left) shows a matched case of an ISC optimization.

Our algorithm considers any statement connected to another statement through a goto as an ISC deopimization candidate. ISC deoptimizations are a wide-catching net for gotos and may not always eliminate gotos. To avoid re-duplicating blocks that other passes condensed, this deoptimization must be run after all other goto-aware deoptimizations. As

a special case, gotos to function-exit blocks, such as a return, are special-cased to reduce duplication. An arbitrary code duplication limit is set on jumps to these types of blocks to mimic the compiler. This allows the `goto cleanup` programming style to still exist in SAILR decompilation.

Reverting condensed statements. After locating all goto edges in the AIL graph, the destination nodes act as the head nodes for duplication. To revert an ISC optimization, the chain of single-successor nodes is copied beginning from the head node. Figure 2.6 (right) shows the effect of the duplication on an ISC case. For a node chain that ends in an exit node, or a node that leaves the current function, it is guaranteed to remove a goto. Listing 4 in the Appendix shows the pseudocode of our ISD deoptimization algorithm.

A common case we observe is that many functions have two exit nodes after a comparison of the stack canary. Therefore, we identify all the exit regions, or clusters of nodes, that end in an exit that has one entry. For stack canary comparison, there are three nodes in the cluster: the if statement, the return node, and the stack-check-fail node. This allows us to copy the entire region as the exit node.

In cases where the final node in the node chain is not an exit node, this deoptimization may not remove the goto. Instead, this deoptimization transforms the graph into a form that is easier to case match for other deoptimizations that run on the graph. This is important because during compilation, ISC optimizations are often stacked on top of other goto-inducing optimizations, similar to the ISD optimizations. SAILR must iteratively revert these optimizations one after another.

2.5.3 Miscellaneous Optimizations

Optimizations not deoptimized by ISD or ISC reverters require specific identification. Of the known miscellaneous optimization, only Switch Lowering may be identified using gotos.

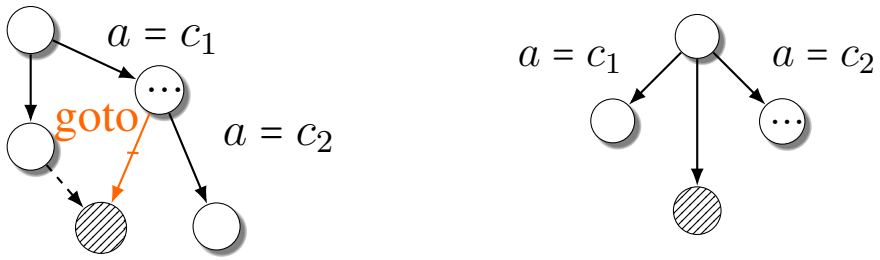

Figure 2.8: CFGs before and after Switch Lowering reversion. All nodes in a candidate switch region must hold a Boolean condition of $a = c$, where c is a constant. These condition nodes must cascade and either be post-dominated by a single node or exit early. A goto appearing in the region makes the case more likely to be a real Switch Lowering case.

Reverting switch lowering. To the best of our knowledge, depending on the density of switch case numbers, GCC may generate three forms of binary control-flow structures for switches: binary decision trees, jump tables, and bit-tests [4]. When fallthroughs between cases are in use, switches that are lowered into decision trees are usually unstructurable as cascading `if-else` constructs. To make such subgraphs structurable, we must transform them into normal switches. In SAILR, we implement a solution based on pattern and condition matching to revert lowered switches. Among all decompilers that we tested, only Hex-Rays has very limited support for reverting lowered switches. In Figure 2.8 a case of cascading `if-else` constructs is shown as well as its transformation into a switch node. A goto present in the cluster is optional but is a high-confidence indicator of a lowered switch.

Reverting switch clustering. Another common cause of switch-related unstructurable subgraphs is switch clustering [4], where a switch is broken into multiple disjoint sub-switches (with different case numbers). Each sub-switch is transformed into its own binary representation (of the three forms). SAILR identifies and merges these sub-switches that are grouped together by conditions within the same region.

2.6 Narrowing Knowledge Gaps

When the decompiler and the compiler diverge in their views of the same binary, it may cause otherwise structurable code to be incorrectly regarded as unstructurable by the decompiler. Therefore, we must close or narrow the knowledge gaps between the decompiler and the original compiler to ensure they have the same view of all structuring-dependent artifacts. Our intuition is that decompilers must be *compiler-aware* to achieve high-quality decompilation. In this section, we summarize two significant sources of knowledge disparity that we identified (described in Section 2.3.3) on GCC-9.5-compiled Coreutils binaries: The returning of functions and missing graph schemas in decompilers.

2.6.1 Non-returning Functions

GCC does not generate edges or successors for function calls (e.g., `exit()`) that do not return. A decompiler that is not aware of these semantics may incorrectly add a spurious edge from the return site (that does not actually exist) to whatever block that follows the call. Unfortunately, fixing this problem requires more than just providing decompilers with a list of non-returning functions, because a function may be returning in some call sites and non-returning in other call sites.

The `error` function from glibc is an example: It returns when the first argument (`status`) is 0, and does not return in other cases. With the `error.h` header file, GCC determines that any call sites for `error` do not return if the first argument has a non-0 constant. To solve these, we augmented the ANGR DECOMPILER so that it determines the returning of `error` call by examining the value of its first argument at all its call sites.

2.6.2 Additional Graph Schemas

Schema-matching algorithms, as described in Section 2.2.3 match subgraphs against known graph schemas that correspond to C structures. By using the methodology outlined in Section 2.3.1, we identified three major missing graph schemas in most decompilers that we tested and added support for them in SAILR.

Short-circuited compound Boolean conditions. During the lifting of C to Gimple, GCC transforms compound Boolean conditions into control-flow constructs using specialized graph schemas to express the short-circuit condition. Both Hex-Rays and Ghidra support such graph schemas, but Phoenix does not and always generates gotos. Moreover, DREAM will generate a semantically equivalent graph where each node is guarded by much more complex conditions than what are in the source. We investigated and concluded that the root cause is DREAM's reliance on condition expression simplification. Simplification (or minimization) of Boolean expressions is an NP-hard problem, so it is too computationally expensive for DREAM to derive Boolean expressions that are as simple as in the source for non-trivial cases (e.g., Boolean expressions with more than six atoms).

Comma-separated multi-statement expressions. A not-that-uncommon usage in C is comma-separated multi-statement expressions, where the statements will be executed first before evaluating the expression at the end. Not supporting this expression type will prevent proper structuring of loops that use such expressions as loop conditions. Out of all decompilers that we test, only Hex-Rays supports such types of expressions. We add support for these expressions in SAILR.

Common condition while-loop exits. In some cases of structuring, the Phoenix algorithm can generate extra gotos that leave cyclic regions structured as `while-loops`. In a `while-loop`, if the loop contains a branch where both successors are inverted conditions of each other, like c and $!c$, this can be replaced by an `if` statement and a break out of the

Table 2.2: Previous work's structuring metrics, GED, and CFGED measured on Figure 2.1 and Listing 1.

	Gotos	LoC	MCC	GED	CFGED
Source	1	19	4	0	0
SAILR	1	15	4	0	0
Phoenix	3	19	4	2	2
DREAM	0	16	9	21	38

loop.

2.7 Measuring the Quality of Structuring

Measuring the structuring quality of decompilation is different from measuring its overall quality. Changes that improve readability, e.g., constant propagation or variable elimination, do not always impact the structure of the code. Previous work focused on evaluating the quality of structuring by relating it to structural complexity. They mainly used three metrics: Number of gotos (Gotos) [50, 107, 17], McCabe Cyclomatic Code Complexity (MCC) [50], and the Lines of Code (LoC) [107]. However, these metrics are flawed for evaluating structuring quality even when shown relatively similar source.

To understand these flaws, we evaluate previous metrics on our motivating example in Listing 1, with the decompilation results by Phoenix, DREAM, and SAILR shown in Figure 2.1 (and measurements produced in Table 2.2). SAILR achieves perfectly structured decompilation, but its MCC is identical to Phoenix's result, despite the latter producing differently structured code from the source. This is because MCC only measures the number of independent paths without considering the resemblance of code structures. As a result, MCC can return significantly lower scores for decompiled code with spurious gotos rather than properly structured conditions because conditions add compounding edges in the CFG. For gotos, although Phoenix's decompilation was structurally closer to the source

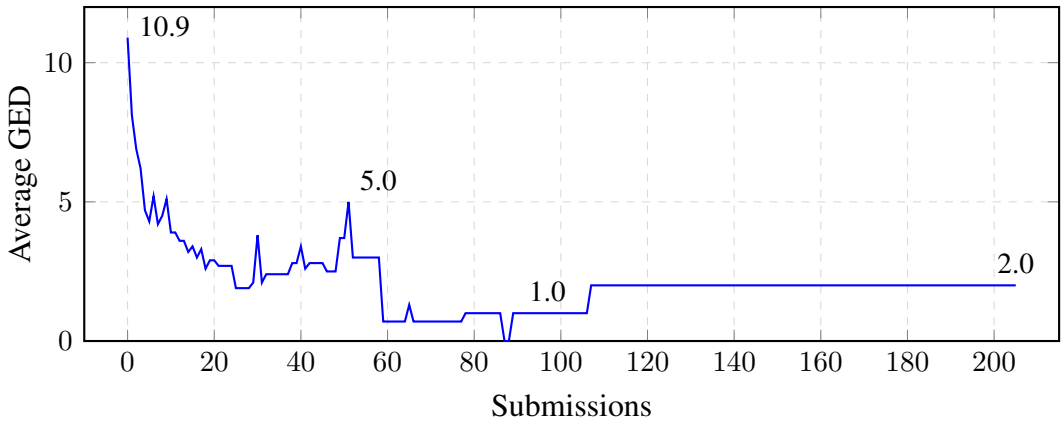

Figure 2.9: In the Decomperson study, a human participant's goal is to perfectly recover the source from only the assembly code of a binary. As human participants submit closer byte-matched decompilation, their average GED scores decrease.

than DREAM's decompilation, Phoenix's score is worse than DREAM's score. Lastly, LoC rewards decompilers for non-structurally related optimizations, such as aggressive expression folding and constant propagation.

2.7.1 Graph Edit Distance

Conceptually, MCC is flawed because it does not consider edge and node locations relative to the source. Using this flaw as inspiration, we used the Graph Edit Distance (GED) relative to the source CFG to encode edge-node location differences into a metric. Prior work has used GED in binary similarity analysis by computing GED on binary-level CFGs [104, 14]. We run GED on the CFGs of the source and associated decompilation to estimate their structural closeness.

We associate lower GED-to-source scores with easier-to-understand decompilation. To study this association, we collected results from Decomperson, a study of human-written decompilation [19]. In Decomperson, participants manually create decompilation given assembly with the target of having the lowest assembly difference from the binary when compiled. With every submission, users attempt to get closer to the source code.

In Figure 2.9, we graph the average exact GED score of 30 participants as they submit

results on the `winky` challenge. We could compute the exact GED scores because the functions in this challenge were sufficiently small. Each participant ended with a byte-match of 50% or higher. Their average GED scores decrease over time, indicating lower GED scores are likely associated with higher closeness to source, which means fewer structural differences.

2.7.2 Control-Flow Graph Edit Distance

Unfortunately, computing an exact GED is NP-hard [18]. In practice, we have found exact GED is usually too expensive to run on graphs with more than 12 nodes. When using upper-bound GED estimation algorithms, their estimations are significantly higher than the exact score, and, on exact match graphs, they rarely return zero. Therefore, we design a scalable algorithm, called Control-Flow Graph Edit Distance (CFGED), that uses CFG-specific characteristics to improve the scalability of GED estimations when applied to decompilation.

CFGED decomposes the GED problem using the identified SESE regions and uses exact GED when the region is small enough, and approximate GED if the region is too large (as the last resort). The CFGED between two CFGs is the sum of the GED of all regions. This often returns a score that is lower than the upper-bound GED approximation in the same or faster time. An example of CFGED computation on two graphs can be found in Appendix 2.12.3.

To validate the accuracy of CFGED, we computed the exact GED, the maximum GED, and the CFGED of all functions with lower than 12 nodes in Coreutils 9.1. On all 506 samples, CFGED returns a score between the maximum and exact. CFGED gets the exact score on 64% of all samples. On average, CFGED introduces a 35% overhead to the percent difference score (the GED approximation divided by the max possible score).

Table 2.3: Structuring results on 7,355 functions across 26 popular Debian packages. The percent change relative to source is shown on each sum. The CFGED percent change is shown w.r.t. Hex-Rays.

Metric	Source			SAILR			Hex-Rays			Ghidra			Phoenix			DREAM			rev.ng		
	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Gotos	1,367	0.19	0.0	2,673 (95.5%)	0.36	0.0	6,115 (347.3%)	0.83	0.0	6,575 (380.9%)	0.89	0.0	8,497 (521.6%)	1.16	0	0 (100%)	0	0	0 (100%)	0	0
Bools	6,180	0.84	0.0	3,980 (35.6%)	0.54	0.0	4,279 (30.8%)	0.58	0.0	4,850 (21.5%)	0.66	0.0	2,685 (56.6%)	0.37	0.0	43,661 (600.5%)	5.94	0.0	2,003 (67.6%)	0.27	0.0
Calls	53,995	7.34	3.0	52,558 (2.6%)	7.15	3.0	52,508 (2.8%)	7.14	3.0	53,202 (1.5%)	7.23	3.0	51,167 (5.2%)	6.96	3.0	51,204 (5.2%)	6.96	3.0	166,798 (116.3%)	22.68	3.0
CFGED	0	0	0	166,468 (0.5%)	22.64	8.0	165,583 (0%)	22.52	8.0	187,509 (13.2%)	25.5	7.0	166,480 (0.5%)	22.64	8.0	338,231 (104.3%)	45.99	10.0	524,248 (216.6%)	71.29	8.0

2.8 Evaluation

We evaluate all structuring algorithms using two sets of C binaries: popular Debian packages and MSVC compilable packages. We attempt to answer the following research questions through experiments:

RQ1. How well does SAILR structure function CFGs during binary decompilation compared to state-of-the-art structuring algorithms?

RQ2. Given that SAILR is guided by findings on GCC 9.5, how does it perform on older and newer GCC versions?

RQ3. Given that SAILR is guided by findings on GCC 9.5, how does it perform on binaries generated by different C compilers?

Additionally, we explore tangentially related questions with an extended evaluation set in Appendix 2.12.4.

2.8.1 Evaluation Methodology

All experiments are performed on an Ubuntu 22.04 server. During each experiment, we compile all .c files in a package and use the compiler to output intermediate source files that have been preprocessed. Next, we compile with debug symbols, no inlining, and saved object files. We then record the address-to-line mapping for each object file before stripping it of debug information. The mapping is recorded for evaluation with CFGED. Each object file has an associated intermediate (.i) file and address mapping.

After compiling, we decompile every object file, which is the non-linked version of the final executable, with every decompiler and parse the output with Joern [108]. We use Joern to collect metrics, as well as CFGs, from each decompiler to allow for decompilation that may not be recompilable. We evaluate Hex-Rays 8.0 (IDA Pro), Ghidra 10.1, and the ANGR DECOMPILER, which implements SAILR, Phoenix, DREAM, and rev.ng's Combing. Each structuring algorithm is evaluated on the metrics in Section 2.8.2. To assure a fair evaluation among all structuring algorithms, we only report metrics on functions that successfully decompiled on all decompilers and did not crash on our measuring pipeline.

Reimplementation of existing algorithms. For fair evaluation, we attempted to use prior work's decompilers. However, the Phoenix decompiler was not open-sourced, and the open-sourced DREAM decompiler does not build in their specified environment. We tried to use the rev.ng decompiler, but the alpha version crashed on most binaries in the Coreutils package, which prohibited us from evaluating it. We reimplemented all of them in the ANGR DECOMPILER.

2.8.2 Metrics

We evaluate the structuring algorithm of every decompiler using two metrics.

Structuredness. Structuredness measures how often a structuring algorithm creates C structures in the output. We follow prior work evaluations [107]: we measure structuredness by counting the number of gotos in the decompilation.

Faithfulness. Faithfulness measures how close the control flow structure is to the original source's. We use CFGED, function calls, and Booleans expressions to measure the similarity in flow, execution structure, and conditions to the source.

We further ensure the structuring correctness of each decompiler's output by manually sampling and verifying a small set of decompiler output. We do not measure the correctness by recompilability because the main contribution of our work is structuring, not recompilable

Table 2.4: Structuring results on 433 functions across Coreutils compiled with various GCC versions and Clang.

	Most-Recent Release	Decompiler	Gotos	Bools	Calls	CFGED
Source	N/A		20	438	4,761	0
GCC 5	October 10, 2017	SAILR	152	295	4,260	14,499
		Hex-Rays	464	299	4,199	14,399
GCC 9	May 27, 2022	SAILR	169	284	4,277	13,462
		Hex-Rays	447	290	4,353	13,266
GCC 11	April 21, 2022	SAILR	170	280	4,290	13,445
		Hex-Rays	451	293	4,353	13,391
Clang 14	March 25, 2022	SAILR	167	292	4,290	22,879
		Hex-Rays	454	303	4,335	22,671

decompilation.

2.8.3 Structuring on Debian Packages

To answer RQ1, we focused on the effects of SAILR concerning programs compiled with GCC 9.5 on the `O2` optimization level, the compiler configuration we studied. We evaluate against all structuring algorithms on 26 popular Debian C packages [1], listed in full in Appendix 2.12.1. Table 2.3 shows the results on each metric for every decompiler.

SAILR outperforms all algorithms in relative goto creation but loses to Hex-Rays in CFGED by 0.5%. Although SAILR loses in CFGED, the high reduction of gotos relative to Hex-Rays indicates SAILR is removing gotos in a non-structurally destructive way. In contrast, DREAM, which removes the most gotos, differs from Hex-Rays by 104.3% on CFGED.

2.8.4 Generalizability Across Compilers

To answer RQ2 and RQ3, we evaluated SAILR on various compiler versions and vendors. Building on the results of Table 2.3, we focus on the differences between the best

Table 2.5: Structuring metrics on 45 unique functions across Zlib compiled with MSVC.

	Gotos	Bools	Calls	CFGED
Source	0	70	158	0
SAILR	12	59	133	1,813
Hex-Rays	14	52	133	1,743

structuring algorithms of the Debian package evaluation: Hex-Rays and SAILR. In Table 2.4 we evaluate multiple versions of GCC as well as Clang on Coreutils, the defacto decompilation evaluation dataset [50, 107, 17]. Although Coreutils has a Windows variant, it does not natively compile with MSVC. In lieu of Coreutils, we chose Zlib since it was easily compilable. In Table 2.5 we evaluate Hex-Rays and SAILR on Zlib, compiled with MSVC 14.20.

Similar to the results in Table 2.3, SAILR marginally loses to Hex-Rays on CFGED (no more than 1.5%), but improves the gotos emitted across every compiler. Because the number of gotos, calls, and Boolean operators are all fairly consistent, we can conclude that the compiler-aware decompilation improvements that SAILR has on GCC 9 actually generalize to historical versions of GCC. However, in the highly different MSVC compiler, SAILR loses significantly more on CFGED, which may be due to hardcoded Windows-specific schemas in Hex-Rays.

2.9 Discussion

To the best of our knowledge, this is the first effort for a complete investigation of goto-inducing compiler transformations and the significance of undoing such transformations in decompilers. We next discuss the limitations of SAILR.

The remaining gotos. While SAILR removes a lot of spurious gotos in decompilation, it does not remove all of them. Through a manual analysis of the remaining gotos, we

find that they are caused by the following reasons: (a) Certain function calls not returning but angr is unaware, which leads to incorrect function-level CFGs; (b) The decompiler choosing the wrong edge among several candidates to virtualize, which prevents SAILR from applying deoptimization techniques at intended locations; and (c) SAILR condensing blocks that were not impacted by ISD optimizations in compilers. We believe (b) and (c) can be addressed by SAILR performing an iterative search when picking edges to virtualize and parts of code to deoptimize, which we leave as future work.

High CFGED between the best decompilation and source. We also investigated why the CFGED scores are still so high, even in cases where SAILR perfectly reverts every goto-inducing optimization that the compiler introduces. There are two main reasons. First, CFGED is still an approximation, which can significantly differ from the actual exact GED on larger CFGs. In a CFG with over 200 nodes and 100 edges, we observed that the exact GED between the decompilation and the source was as low as 4 while CFGED reported 306. We envision that future algorithmic improvements on GED computation will reduce GED scores in our measurement. Second, CFGED is extremely sensitive to structural differences, and many structuring choices that a decompiler makes (e.g., whether a node belongs to the loop body or not), while leading to high structural differences, are ultimately undecidable. We believe the use of statistical techniques, e.g., machine learning, will help immensely in these cases.

Architecture, platform, and compiler-specific study. Our investigation is specific to x86-64 Linux user-space executables that are compiled using GCC or Clang, and some MSVC-compiled Windows binaries. While it is likely to find new types of goto-inducing compiler transformations under other settings, decompiler authors can use the methodology we presented in Section 2.3.2 to find and address them.

Compounding ISC and ISD optimizations do not always cause gotos. In rare cases, compounding ISC and ISD optimizations may create subgraphs that happen to match

against known graph schemas. As such, no unstructurable code regions will be created. A graph-schema-matching decompiler will generate a goto-free decompilation result, however it does not structurally match the source. Fundamentally, this is caused by the many-to-one mapping from legitimate C source to binary code. We believe a natural solution is using a machine-learning approach to predict the most suitable structuring result from the binary CFG and its context.

Using Hex-Rays to study gotos in Coreutils binaries. In the goto-origin study, we used Hex-Rays to investigate gotos in binary code with the assumption that Hex-Rays only performs basic graph-schema-matching and does not proactively remove gotos. We found that this assumption is false because Hex-Rays seems to have more graph schemas than other decompilers. Fortunately, we did not notice any cases where Hex-Rays would actively solve ISC- or ISD-optimizations. We believe using a naive graph-schema-matching decompiler can help increase accuracy of all goto sources.

2.10 Related Work

Control-flow Structuring. The two main methods for recovering control-flow structures from CFGs are *interval analysis* and *structural analysis*. Interval analysis [5, 25] partitions a CFG into nested regions called intervals. The nested nature ensures the data-flow analysis is not duplicating effort.

Structural analysis [93] is an extension of interval analysis, allowing a syntax-driven method of data-flow analysis that is normally reserved for abstract syntax trees to be applied on intermediate representations of low-level code such as AIL, VEX, and BIL. Structural analysis algorithms recover high-level control constructs from CFGs for use in decompilation.

Engel et al. [36] extended structural analysis to C-specific control statements, by proposing the SESS model. SESS accounts for statements in C that appear before control-flow

changes induced by a `break` or `continue`. However, this approach relies on pre-defined schemas that occur in CFGs and introduces gotos that were not originally present in the source code. The SAILR approach focuses specifically on GCC-induced *gotos*.

A related area of research attempts to remove gotos at the source code level [38, 102] by defining AST transformations that replace gotos with an equivalent semantic-preserving structure. This generally increases overall code size and complexity and also can insert unnecessary Boolean variables.

Decompilers. Cifuentes' PhD thesis [22] can be considered the academic foundation for many modern decompilers. The decompilation techniques, based on interval theory and expanded on in follow-up work [23], were implemented in the Intel 80286/DOS to C decompiler `dcc`.

Hex-Rays, a plugin for IDA, is the de facto commercial decompiler. There is no official source on the techniques Hex-Rays uses for decompilation. Schwartz et al. [17] noted that Hex-Rays uses some form of improved structural analysis.

The Phoenix [17] decompiler is built with the Binary Analysis Platform (BAP) [16]. Phoenix attempts to reduce the number of gotos in decompilation through *iterative refinement* which will replace an edge with a goto if the structural analysis algorithm cannot make forward progress. DREAM [107] builds upon the techniques in the Phoenix [17] decompiler to emit zero gotos by design. DREAM forcefully excludes gotos with the use of Boolean expressions and state variables. While it eliminates gotos, it can generate code with entangled execution paths [50]. rev.ng is a decompiler [50, 30] implementing the *Control Flow Combing* algorithm to remove gotos by aggressively duplicating code. These decompilers treat the symptom of the problem—gotos appearing in decompilation—but do not address the root of the problem. SAILR attempts to remedy this by following a simple idea: to achieve high-quality decompilation, decompilers must be compiler-aware.

Recent advances in deep learning techniques have led to progress in the field of de-

compilation. Researchers have used deep learning approaches to (1) enhance the quality of decompilation by predicting debug information [54], variable names [58, 64, 20, 21], types [20, 114] and function names [8], and (2) build an end-to-end, neural network-based decompiler [44]. These approaches open up exciting opportunities for future research aimed at developing a general-purpose end-to-end neural decompiler.

2.11 Conclusion

Decompilers seek the impossible—how to recover source code using only binary code. This act of divination requires a complex mix of binary analysis, graph analysis, and, fundamentally, engineering. We believe that the goal of a decompiler should be to get as close as possible to the original code and that the only way to accomplish this is to be compiler-aware. SAILR and the ANGR DECOMPILER represent the next steps in this direction toward perfect decompilation.

2.12 SAILR Appendix

2.12.1 Popular Debian Packages for Evaluation

We evaluated SAILR on 26 randomly selected from the top 50 Debian packages collected from the Popcon [1] list: bash, libselinux, shadow, libedit, base-passwd, openssh-portable, dpkg, dash, grep, kmod, diffutils, findutils, gnutls, iproute2, gzip, sysvinit, bzip2, libacl, libexpat, libbsd, tar, rsyslog, cronie, zlib, e2fsprogs, and coreutils.

2.12.2 Case Study: Constant Depropagation

To better understand more complex cases of deoptimization, we document a case study involving constant depropagation in the Coreutils 9.1 package of compounding optimizations.

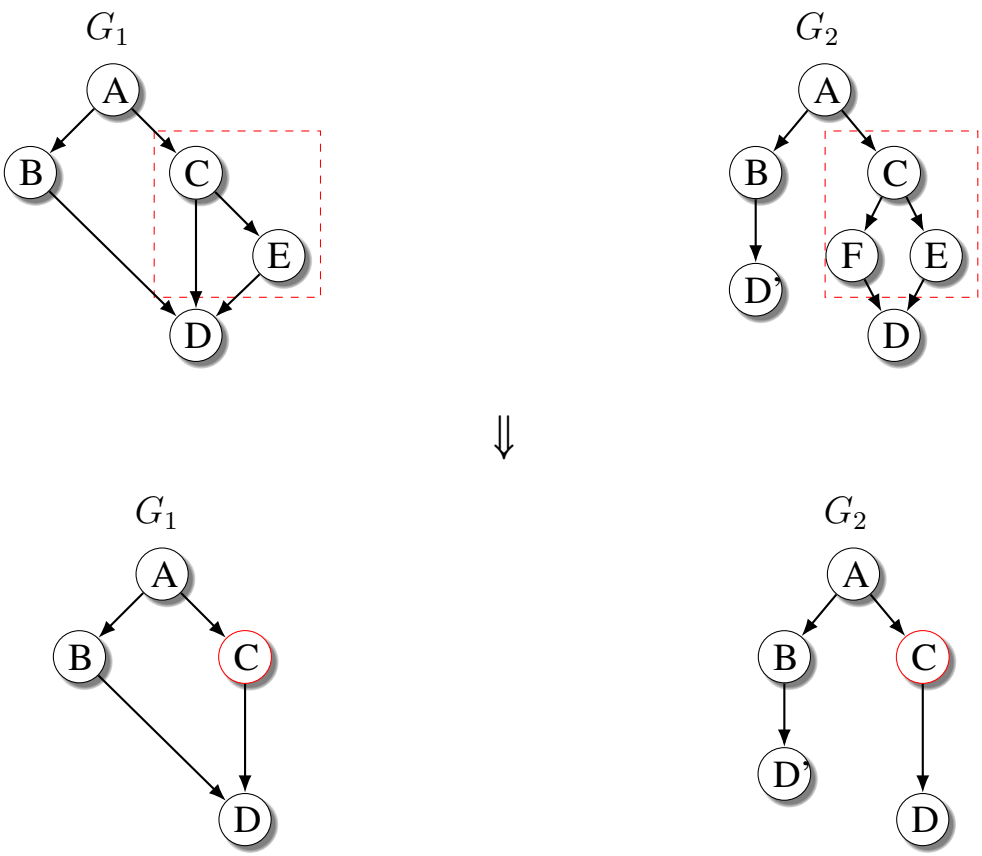

Figure 2.10: A single round of CFGED approximation and region collapsing. Every round, a region collapses after computing the GED inside the region. Additionally, the difference in the edges source and destination are added as edits.

The binary `fmt` contains an ISD optimization in the function `main`, which causes a duplication of a subgraph shown in Listing 2. However, this case does not match any ISD deoptimization schema because the calls to `xdectoumax` are not the same, differing by a single argument value. The calls on Lines 7 and 11 originate from the same source line and are duplicates, but the constant which is assigned to `max_width` on Line 2 is constant-propagated, causing them to differ after the ISD optimization. To revert this ISD case, the optimizations must be undone in reverse order.

SAILR first runs a KMP similarity matching algorithm to find arguments that may differ across calls. When these arguments are identified, SAILR checks to see if one argument is symbolic (such as a variable) and one is constant. If this is true, SAILR uses

Reaching Definitions Analysis to find all the possible values of the symbolic argument on the path with the constant. If the symbolic argument can only equal the constant on that path, it is replaced by the symbolic variable.

In Listing 2 this makes both calls to `xdectoumax` an exact duplicate. Because both calls are duplicates, have a common successor to their blocks, and at least one is connected by a goto, this case matches the ISD algorithm. When this case is merged only one `xdectoumax` remains, with the assignment blocked by the reaching conditions of the two statements.

2.12.3 Computing CFGED

Figure 2.10 shows an example computation of CFGED between G_1 and G_2. CFGED starts by finding two tail regions that contain the same head node across graphs. In the decompilation-to-source comparison, the addresses associated with each decompilation line are used to map source lines to nodes. Because a tail region contains no nested regions, the size of these sub-graphs is often small when mapped correctly.

The highlighted regions in Figure 2.10 have a GED of 2. Because there are also edges leaving these regions, we must check the edit distance of the region's successor. There is a missing edge and node in G1 that should have an edge to the successor. Adding this node and edge constitutes an edit distance of 2. In practice, this can be computed by running GED on the region plus its successor node. In this example, round one has a total CFGED score of 4: 2 for the single edge and node conflict (F, D) and 2 for the GED of the regions.

Continuing, the next region collapsing round would have no more regions to collapse and would run GED on G_1 and G_2. The resulting GED is 3, with the total CFGED to *7*, which is equal to the exact GED score of 7.

```
max_width = 0x4b;
if (v0) {
  max_width = xdectoumax(v0, 0x0, 0x9c4, &.LC11,
      dcgettext(NULL, "invalid width", 0x5), 0x0);
  if (!v1)
      goto LABEL_40cd86;
  goal_width = xdectoumax(v1, 0x0, max_width, &.LC11,
      dcgettext(NULL, "invalid width", 0x5), 0x0);
} else {
  if (v1) {
    goal_width = xdectoumax(v1, 0x0, 0x4b, &.LC11,
        dcgettext(NULL, "invalid width", 0x5), 0x0);
    max_width = goal_width + 10;
    goto LABEL_40ccc7;
  }
Label_40cd86:
  goal_width = (max_width * 187 >> 31
                CONCAT max_width * 187) /m 200;
}
Label_40ccc7:
v17 = optind;
```

Listing 2: An example ISD case from the Coreutils binary fmt on function main.

2.12.4 Extended Evaluations

Ablation Study. In this experiment, we measure the impact of deoptimization (discussed in Section 2.5) and knowledge expansion (Section 2.6). We use three structuring algorithm variants: Phoenix with improvements in Section 2.6.1, SAILR Lite (Phoenix with improvements in Section 2.6), and SAILR. We decompile 953 functions in Coreutils 9.1 compiled under optimization level `O2`. As shown in Table 2.6, both deoptimization and knowledge expansion are necessary for reducing gotos and CFGED scores. Deoptimization contributes more than knowledge expansion in terms of goto reduction.

Linux Kernel Study. To better understand how SAILR structures code highly saturated

Table 2.6: Results of the ablation study on Coreutils investigating the impact of deoptimization and knowledge expansion.

	Gotos			Bools			Calls			CFGED		
	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Source	40	5	0	1,291	27	1.35	11,690	173	12.27	0	0	0
Phoenix	1,761	37	1.85	534	12	0.56	10,136	173	10.64	40,733	918	42.74
SAILR Lite	1,410	67	1.48	932	20	0.98	10,063	173	10.56	40,384	928	42.38
SAILR	666	19	0.7	770	20	0.81	10,322	173	10.83	40,489	1,382	42.49

Table 2.7: Linux kernel structuring evaluation.

	Source	SAILR	Hex-Rays	Ghidra	Phoenix	DREAM	rev.ng
Gotos	120	136	311	352	457	0	0
Bools	737	345	354	519	277	2,759	150
Calls	4,306	3,668	3,376	3,348	3,226	3,248	5,347
CFGED	0	19,096	18,916	18,576	18,956	29,400	25,903

with gotos, we evaluate SAILR against other structuring algorithms on a subset of the Linux kernel [62]. We randomly picked 100 source files (and object files) from the core kernel code base, collected 802 functions, and ran decompilers against them. Table 2.7 shows the result: SAILR outperforms other solutions in terms of relative goto emittance to source but shows slightly worse CFGED scores.

Malware Study. We also evaluated SAILR on a malware sample: the SpyEye malware on Windows [97]. In Table 2.8, we compare SAILR against other structuring algorithms on 54 functions. Because there is no source code to compare against, we use generic readability

Table 2.8: Decompilation of SpyEye with generic readability metrics.

	SAILR	Hex-Rays	Ghidra	Phoenix	DREAM	rev.ng
Gotos	0	0	0	20	0	0
Bools	17	15	17	8	33	1
LoC	1,640	878	1,881	1,690	1,681	2,186

metrics, which may not reflect structural closeness.

Decompilation Times. In Table 2.9 we show the decompilation times across 1272 functions in Coreutils 9.1 compiled with GCC 9.5 `O2`. The ANGR DECOMPILER takes significantly longer to decompile due to its Python implementation. SAILR takes longer than our other structuring algorithms because of a naive implementation for the search phase for ISD deoptimization. In short, SAILR uses a shortest-path algorithm to find the head for deduplication, which we believe can eliminated with future engineering.

Table 2.9: Decompilation times (in seconds) across all functions in Coreutils.

	SAILR	Hex-Rays	Ghidra	Phoenix	DREAM	rev.ng
Median	1.16	0.01	0.05	0.82	0.8	0.71
Mean	5.24	0.03	0.07	2.15	2.08	2.33
Max	606.7	0.38	1.06	55.32	31.55	329.33
Sum	6,670.0	31.84	91.43	2,735.03	2,647.14	2,962.65

Compiler Configurations Study. To understand the full effect of different compiler optimizations on SAILR, we compiled the same 26 Debian packages in Section 2.8.3 on all optimization levels for GCC 9.5. In Table 2.10 we show a comparison against other structuring algorithms on 5,752 functions. The result shows that SAILR consistently yields the closest number of gotos (except `O0`) with CFGED scores being very close (within 6%) to the lowest CFGED scores (from Hex-Rays). We believe SAILR's poor results on gotos and CFGED compared to Hex-Rays is due to Hex-Rays maturity in schema for unoptimized binaries.

Expanded Multi-Compiler Study. We expanded our results from Table 2.4 in Table 2.11 to include all decompilers supported in our evaluation pipeline. Note, that there are fewer functions (392 total) in these results due to functions missing across more decompilers.

```
def deoptimize_isd(duplicate_graph_pair, graph):
    merged_graph = merge_pair_graphs(duplicate_graph_pair)
    remove_pair_nodes_from_graph(graph_pair, graph)
    graph = compose_graphs_of_many(graph, merged_graph)
    for head_node in heads_of_graphs(merged_graph):
        point_old_predecessors_to_node(head_head)
    for leaf_node in ends_of_graphs(merged_graph):
        conditional_node = make_reaching_cond_node(leaf_node)
        graph.add_edge(leaf_node, conditional_node)
    for node in new_conditional_nodes(graph):
        original_leafs = leaf_nodes(duplicate_graph_pair)
        for leaf in original_leafs:
            graph.add_edge(
                node,
                original_edge_by_condition(leaf)
            )
    for node in new_conditional_nodes(graph):
        simplify_and_merge_adjacent_conditionals(node)
    return graph
```

Listing 3: Pseudocode of the SAILR ISD deoptimization algorithm. It takes as input a pair of correctly identified duplicate graph pairs and the original graph to be deoptimized. It outputs the deoptimized graph.

Table 2.10: A comparison of structuring algorithms across optimization levels on GCC 9.5 for Debian packages.

	Metric	O0			O1			O2			O3		
		Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Source	Gotos	1,000	0.17	0.0	1,000	0.17	0.0	1,000	0.17	0.0	1,000	0.17	0.0
	Bools	4,252	0.74	0.0	4,256	0.74	0.0	4,256	0.74	0.0	4,256	0.74	0.0
	Calls	38,828	6.75	3.0	38,897	6.76	3.0	38,897	6.76	3.0	38,897	6.76	3.0
	CFGED	0	0	0.0	0	0	0.0	0	0	0.0	0	0	0.0
SAILR	Gotos	426	0.07	0.0	1,148	0.2	0.0	1,542	0.27	0.0	1,596	0.28	0.0
	Bools	3,382	0.59	0.0	2,628	0.46	0.0	2,651	0.46	0.0	2,731	0.47	0.0
	Calls	38,021	6.61	3.0	38,818	6.75	3.0	37,921	6.59	3.0	38,221	6.64	3.0
	CFGED	92,198	16.03	5.0	105,786	18.39	7.0	110,358	19.19	7.0	115,517	20.08	7.0
Hex-Rays	Gotos	652	0.11	0.0	2,654	0.46	0.0	3,718	0.65	0.0	3,924	0.68	0.0
	Bools	4,416	0.77	0.0	2,935	0.51	0.0	2,936	0.51	0.0	3,046	0.53	0.0
	Calls	39,433	6.86	3.0	39,250	6.82	3.0	38,037	6.61	3.0	38,311	6.66	3.0
	CFGED	86,658	15.07	4.0	104,118	18.1	7.0	109,028	18.95	7.0	114,335	19.88	7.0
Ghidra	Gotos	1,453	0.25	0.0	3,311	0.58	0.0	4,209	0.73	0.0	4,324	0.75	0.0
	Bools	4,341	0.75	0.0	3,270	0.57	0.0	3,366	0.59	0.0	3,741	0.65	0.0
	Calls	39,305	6.83	3.0	40,203	6.99	3.0	38,767	6.74	3.0	38,960	6.77	3.0
	CFGED	117,150	20.37	5.0	123,810	21.52	7.0	125,989	21.9	7.0	130,729	22.73	7.0
Phoenix	Gotos	3,767	0.65	0.0	4,438	0.77	0.0	5,501	0.96	0.0	5,712	0.99	0.0
	Bools	2,631	0.46	0.0	1,803	0.31	0.0	1,878	0.33	0.0	1,960	0.34	0.0
	Calls	38,034	6.61	3.0	38,000	6.61	3.0	37,062	6.44	3.0	37,303	6.49	3.0
	CFGED	94,257	16.39	4.0	106,807	18.57	7.0	110,871	19.28	7.0	115,350	20.05	7.0
DREAM	Gotos	0	0	0	0	0	0	0	0	0	0	0	0
	Bools	10,976	1.91	0.0	17,448	3.03	0.0	28,654	4.98	0.0	30,048	5.22	0.0
	Calls	38,036	6.61	3.0	38,023	6.61	3.0	37,090	6.45	3.0	37,337	6.49	3.0
	CFGED	138,318	24.05	5.0	176,340	30.66	8.0	223,428	38.84	9.0	234,183	40.71	9.0
rev.ng	Gotos	0	0	0	0	0	0	0	0	0	0	0	0
	Bools	1,732	0.3	0.0	1,747	0.3	0.0	960	0.17	0.0	974	0.17	0.0
	Calls	93,713	16.29	3.0	108,128	18.8	3.0	97,076	16.88	3.0	97,752	16.99	3.0
	CFGED	271,168	47.14	6.0	292,823	50.91	7.0	282,414	49.1	7.0	295,824	51.43	7.0

Table 2.11: A comparison of decompilation on Coreutils binaries compiled by GCC 5, GCC 9, GCC 11, and Clang 14 with O2.

	Metric	GCC 5			GCC 9			GCC 11			Clang 14		
		Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Source	Gotos	4	0.01	0.0	4	0.01	0.0	4	0.01	0.0	4	0.01	0.0
	Bools	273	0.7	0.0	263	0.67	0.0	263	0.67	0.0	266	0.68	0.0
	Calls	3,724	9.5	5.0	3,726	9.51	5.0	3,726	9.51	5.0	3,781	9.65	5.0
	CFGED	0	0	0	0	0	0	0	0	0	0	0	0
SAILR	Gotos	56	0.14	0.0	68	0.17	0.0	70	0.18	0.0	66	0.17	0.0
	Bools	168	0.43	0.0	159	0.41	0.0	160	0.41	0.0	172	0.44	0.0
	Calls	3,418	8.72	4.0	3,430	8.75	4.0	3,420	8.72	4.0	3,431	8.75	4.0
	CFGED	11,880	30.31	7.5	8,209	20.94	8.0	8,272	21.1	7.5	9,027	23.03	9.0
Hex-Rays	Gotos	208	0.53	0.0	226	0.58	0.0	224	0.57	0.0	216	0.55	0.0
	Bools	184	0.47	0.0	179	0.46	0.0	177	0.45	0.0	180	0.46	0.0
	Calls	3,485	8.89	5.0	3,485	8.89	5.0	3,487	8.9	5.0	3,405	8.69	4.0
	CFGED	11,727	29.92	7.0	8,065	20.57	7.0	8,041	20.51	7.0	8,966	22.87	9.0
Ghidra	Gotos	216	0.55	0.0	215	0.55	0.0	203	0.52	0.0	101	0.26	0.0
	Bools	208	0.53	0.0	203	0.52	0.0	206	0.53	0.0	228	0.58	0.0
	Calls	3,525	8.99	5.0	3,579	9.13	5.0	3,577	9.12	5.0	1,846	4.71	2.0
	CFGED	12,373	31.56	7.0	8,878	22.65	7.5	8,810	22.47	7.0	8,859	22.6	7.0
Phoenix	Gotos	291	0.74	0.0	306	0.78	0.0	300	0.77	0.0	339	0.86	0.0
	Bools	111	0.28	0.0	115	0.29	0.0	113	0.29	0.0	115	0.29	0.0
	Calls	3,397	8.67	4.0	3,399	8.67	4.0	3,400	8.67	4.0	3,403	8.68	4.0
	CFGED	11,966	30.53	7.0	8,370	21.35	8.0	8,434	21.52	7.5	9,357	23.87	9.0
DREAM	Gotos	0	0	0.0	0	0	0.0	0	0	0.0	0	0	0.0
	Bools	1,569	4.0	0.0	1,640	4.18	0.0	1,774	4.53	0.0	1,411	3.6	0.0
	Calls	3,397	8.67	4.0	3,400	8.67	4.0	3,401	8.68	4.0	3,402	8.68	4.0
	CFGED	16,940	43.21	9.0	13,476	34.38	9.5	13,389	34.16	9.0	14,245	36.34	12.0
rev.ng	Gotos	0	0	0.0	0	0	0.0	0	0	0.0	0	0	0.0
	Bools	29	0.07	0.0	33	0.08	0.0	32	0.08	0.0	72	0.18	0.0
	Calls	5,228	13.34	5.0	5,284	13.48	5.0	5,235	13.35	5.0	5,738	14.64	5.0
	CFGED	17,457	44.53	7.0	13,289	33.9	8.0	13,447	34.3	8.0	17,066	43.54	10

```
def deoptimize_isc(target_node, graph):
    exit_regions = get_exit_regions(graph)
    node_chain = []
    next_node = graph.successors(target_node)[0]
    graph.remove_edge(target_node, next_node)
    while next_node is not None:
        node_chain.append(next_node)
        successors = graph.successors(next_node)
        if len(successors) > 1:
            break
        next_node = successors[0]
    last_node = target_node
    for node in node_chain
        if is_region_head(node, exit_regions):
            new_nodes = copy_region_nodes(node)
            graph.add_edges(last_node, new_nodes)
            break
        else:
            graph.add_edge(last_node, copy(node))
            last_node = node
    return graph
```

Listing 4: Pseudocode of the SAILR ISC deoptimization algorithm.

Chapter 3

STUDYING THE REVERSE ENGINEERING PROCESS

3.1 Introduction

The first step to securing software is understanding it; after all, complex tasks such as finding unintended program behavior first require comprehending its intended behavior. In the modern age, programs are developed at breakneck speeds, requiring an understanding of more software than ever. To understand this software, often foreign to its analyst, is known as Software Reverse Engineering (SRE).

Unfortunately, SRE is a complex and primarily human-driven process [71, 101, 85]. SRE practitioners use a variety of strategies to understand software, frequently carrying out multiple subtasks in the process [71]. These subtasks are typically performed iteratively, and the relationships among them are often complex [71, 101].

Recently, LLMs have shown promise for approaching largely human-driven tasks in security [87, 92]. As such, rapid work by both the research and practitioner communities explores integrating LLMs into SRE tasks. Researchers have approached isolated tasks, such as symbol recovery [56, 59, 103, 105], type inference [56, 103], and vulnerability identification [29, 67]. In response, practitioners integrate these initial findings into binary decompilers, e.g., Hex-Rays Decompiler (ships with IDA Pro) and Ghidra, among other industry-standard SRE tools [57, 99, 9, 39]. However, despite their known advancements in individual tasks, no work exists to show whether these LLMs benefit the SRE process from the human perspective.

Similar to human-AI teaming [111], the interaction between an SRE practitioner and an LLM impacts the performance of LLM-enhanced, human-driven SRE. We refer to this

interaction and LLM results interpretation as the *dynamics* between SRE practitioners and LLMs. Existing studies, which focus on studying LLM improvements to SRE tasks in isolation [56, 59, 103, 105, 92], fail to consider the human-LLM dynamics involving iterative sub-tasks. These studies do not directly or adequately measure the impacts that LLMs have on the entire human-driven SRE process. This represents a lost opportunity to optimize LLM-enhanced, human-driven SRE.

In this work, we present the first study of the dynamics between humans and LLMs during SRE. To accomplish this systematic study, we present three distinct phases. First, to understand the current LLM-SRE community, we survey 153 SRE practitioners who have used LLMs in SRE. We discover that practitioners rely on six distinct features to accomplish SRE tasks. Using these results, with the help of 13 experts in SRE, we design a fine-grained human study involving two CTF-style challenges, a decompiler plugin to support the six discovered features, and an online platform for instrumentation. Finally, we conducted this study on 48 participants, relying on both quantitative and qualitative data found in observations, writeups, and post-study responses.

In our study, we observe over 6,586.0 minutes of SRE across 24 experts and 24 novices. Participants interacted with LLMs a total of 1,517 distinct times, across two challenges, on over 50 functions. Through an analysis of their behavior, we arrive at 18 distinct findings on the impacts and best strategies for LLM use in SRE. These findings enable us to answer *where*, *when*, and *how* LLMs can augment the SRE process. We find that they have both benefits and harms to SRE.

We find LLMs shorten the skill gap for novice SRE practitioners, improving their performance by 98%, helping them approach expert levels of SRE. On average, LLMs reduce the analysis time of standard algorithms by up to 2.4× and enable users to recover more artifacts lost to compilation (at least 66% more). However, these benefits were not without their harms. Experts showed nearly no change in performance when utilizing an LLM,

and in some cases, were harmed by their hallucinations, namely on vulnerability identification. Artifacts recovered by LLMs, more frequently than not, were unhelpful, causing more noise in the SRE process. Additionally, novel and large code continues to degrade LLMs' quality, hurting reverse engineers who overrely on them.

We observe, as it stands, that LLMs play an assistant role in SRE. They are best utilized when they act as a quick and under-relied-upon filter for understanding, not a replacement. LLMs show promise in being the first line of analysis, but expertise is still required to interpret the results.

Contributions. This work makes the following contributions:

- We create a snapshot of the state of LLM-SRE between 2024 and 2025, and establish six ways SRE practitioners rely upon and use LLMs.
- We illuminate 18 findings surrounding the integration of LLMs into the SRE process, including their effects, their best uses, and how they might be improved upon.
- We implement and open source a platform [1] and a decompiler plugin [2] for studying fine-grained human-driven SRE with LLMs in decompilers, including both behavior instrumentation and practitioner-utilized LLM features.

Our work takes the first steps toward understanding and optimizing how humans and LLMs collaborate for SRE. While the results are promising, they are constrained by some limitations: the scale and representativeness of our study challenges, the restricted tooling (limited to static analyses and excluding dynamic or advanced non-LLM tools), the absence of participant assessments of SRE environment satisfaction, and the reliance on self-reported expertise. These factors may limit generalizability (Section 3.10).

[1] https://github.com/mahaloz/dec-synergy-study

[2] https://github.com/mahaloz/DAILA

However, even if our work is not definitive, it establishes a foundation for future research on human-AI collaboration in SRE. Our findings will help the community explore impactful ways to enhance human-LLM collaboration in SRE. It also offers insights for education, future research, and practical tool development for real-world reverse engineering.

3.2 Background and Scope of the Study

This work explores the intersection of two domains: Software Reverse Engineering and Large Language Models.

Software Reverse Engineering (SRE) involves analyzing software to uncover its design and functionality. Applications range from malware analysis to vulnerability discovery and piracy prevention. The primary goal is to reconstruct program logic and identify conditions that trigger specific code behaviors, often linked to bugs or malicious actions. SRE typically unfolds in multiple phases and involves tools like IDA Pro and Ghidra, which integrate disassembly, decompilation, and debugging in a unified interface. Both static and dynamic analyses are employed: the former inspects code without execution (e.g., file structure, functions, assembly), while the latter monitors runtime interactions with memory and the OS.

This study focuses on *static binary analysis*, examining executables without execution. Our participants interact with disassembled and decompiled code via IDA Pro, aligning with traditional program comprehension research [13, 71] that emphasizes reading over debugging. We extend traditional research in this area by allowing access to decompilation, which is commonly used in SRE, as it simplifies binaries for understanding [106, 10].

Large Language Models (LLMs) are transformer-based neural networks trained on vast text corpora using self- and semi-supervised methods [89, 100]. Modern LLMs, typically using decoder-only architectures [98], excel at general-purpose language generation. Initially, task adaptation required fine-tuning. Today, *prompt engineering* guides model be-

havior through carefully designed input prompts, leveraging pre-trained knowledge without retraining [15]. This has proven efficient and often comparable to fine-tuning for large models.

Scope of the study. We aim to investigate how LLMs are being integrated into the SRE workflow, specifically for static code understanding, and assess whether their use enhances analysts' performance compared to traditional methods. Through this study, we intend to answer the following research questions:

RQ1: How do SRE practitioners integrate LLMs into the SRE process, and what are their perceptions? (Section 3.4)

RQ2: How does the inclusion of LLMs in the SRE process impact the performance of practitioners? (Section 3.6)

RQ3: How do practitioners interact with LLMs, and what factors influence their interactions? (Section 3.7)

3.3 Methodology

In this section, we describe the three phases of our methodology, discuss the recruitment process, and present the statistical methods employed throughout our study.

3.3.1 Three Study Phases

To answer our research questions, we systematically explore, design, and complete our study in three phases.

I) Formative research (Section 3.4). We conduct both a comprehensive literature review (including practitioner tools) and an online pre-study survey to understand the design requirements for our study. This phase informs our design on how practitioners interact with LLMs during the SRE process and through what platforms. In total, we use 153 survey responses to create a snapshot of modern LLM for SRE use between 2024 and 2025.

II) Study design (Section 3.5). Next, we design the human study and platform for measuring LLM use in SRE. We use the Phase I results for both the SRE platform (IDA Pro) and LLM features used in our study. We then design two binary executable CTF-style challenges for practitioners to complete, with a writeup required for explaining each solution. We design the challenges and platform iteratively with a team of 13 SRE experts to be representative of real-world difficulties that practitioners may face. Finally, we design a post-study survey to qualitatively confirm the quantitative results we find in the experiment.
III) Empirical Study and Analysis (Section 3.6 and Section 3.7). Using the design from Phase II, we run our study on 48 participants (a subset of the 153 survey respondents in Phase I) and analyze the results. We analyze three sources of data in this phase: 1. fine-grained quantitative data, such as clicks and function renames actioned during the study, 2. solution writeups used to assess participants' understanding of the challenges, and 3. free responses in the post-study survey capturing general sentiments of LLM usefulness. We analyze the results with two goals. First, we analyze how LLMs impacted the SRE process, from the understanding level to the solving speed. Second, we explore how participants extracted useful responses from LLMs with different strategies.

3.3.2 Participant Recruitment

We recruited all participants online during 2024 and 2025. We primarily recruited participants through personal and professional connections to ensure that they have prior experience using LLMs for SRE. Recruits include students with relevant experience (e.g., taking malware analysis courses), researchers from eight universities, and experts from six renowned cybersecurity companies with headquarters located on different continents.

We do not offer compensation for the anonymous pre-study survey, which complies with prior work practices [71, 6]. Participants who expressed interest in Phase III provided their email addresses for contact. Contacted participants who completed the SRE session

were compensated with a $50 gift card. This includes participants who *finished* both challenges, but may not have solved both correctly.

3.3.3 Statistical Analysis of Quantitative Data

We implement a rigorous methodology that integrates statistical testing and effect size estimation in our study. Because the field of human reverse engineering study is emerging, there are no established effect size thresholds specific to SRE tasks. Thus, we situate our methodology within the broader domain of software engineering, which also involves code comprehension, a key component of our participants' activities. We adopt the guidelines proposed by Kampenes et al. [61] and conduct all statistical tests using a significance threshold of $\alpha = 0.05$. In cases involving multiple comparisons, we apply the Benjamini-Hochberg correction [11] to control the false discovery rate (FDR).

Comparing two independent samples. For any given challenge, a participant is assigned to either the control or treatment group independently of their assignment on the other challenge. With each participant completing two challenges, we randomly assign their treatment group (LLM support) to one of the two challenges. This design, together with our decision to make two vastly different challenges, ensures that prior exposure to one challenge does not influence the performance of the other, which addresses the potential carryover effect.

Statistical methodology. I) We assess the normality of the data using the Shapiro-Wilk test. II) If the data is consistent with normality, we assess the equality of variances using Levene's test, with the mean as the center parameter for symmetric distributions [46]. III) If Levene's test fails to reject the null hypothesis, we assume equal variances and use the two-sample (pooled) t-test. Otherwise, if it indicates significant variance differences, we use Welch's t-test. IV) In case of statistically significant difference, we computed Cohen's *d* to quantify the magnitude; for samples with $n < 20$, we applied Hedges' g adjustment, which

adds a small bias correction [26, 55, 73]. V) In order to estimate the effect size, we adopted the finer-grained classification of Funder et al. [45], who further refined Cohen's guidelines [27] that are frequently employed in social sciences and software engineering [7]. Accordingly, we adopted the following interpretation: effect size es is classified as small ($es \leq 0.1$), medium ($0.1 < es \leq 0.2$), large ($0.2 < es \leq 0.8$), and very large ($es > 0.8$).

However, if the assumption of normality is violated (point I) or the sample size is too small ($n < 11$), we follow the recommendations of Fay et al. [41] and Gibbons et al. [47]. Namely, we employ the Mann-Whitney U test as a robust nonparametric alternative to the independent samples t-test. Given that the literature suggests caution with very small samples [40, 53], we never test with fewer than five observations per group. We also compute Cliff's Delta δ as a robust nonparametric effect-size measure [24, 49]. It ranges from $-1 \leq \delta \leq 1$, and measures how often values in one distribution tend to be larger than values in another distribution. We adopt commonly used Cliff's δ thresholds in software engineering [7]: negligible ($\delta < 0.147$), small ($0.147 \leq \delta < 0.33$), medium ($0.33 \leq \delta < 0.474$), and large ($\delta \geq 0.474$).

Correlation. To assess the presence of a statistically significant relationship between two variables (i.e., equal size), we implement a flexible correlation testing procedure that selects the appropriate statistical test based on the distributional properties of the input data, enabling us to capture both linear and non-linear associations without violating statistical assumptions. If both datasets are found to be normally distributed (using the Shapiro-Wilk test), we compute the Pearson correlation coefficient, which reflects the strength and direction of a linear relationship. Conversely, if one or both groups deviate from normality, we use Spearman's rank-order correlation, which is more appropriate for ordinal data or non-linear monotonic relationships. The effect size is again interpreted in accordance with Funder et al. [45].

3.4 Formative Research

This section investigates **RQ1:** *How do SRE practitioners integrate LLMs into the SRE process, and what are their perceptions?*

We first investigate how practitioners use and integrate LLMs into the SRE process to guide our experiment design in Section 3.5. Then, we design an online survey to provide us with a road map for accurately studying how practitioners use LLMs. All questions are motivated by a literature review of recent works in LLMs for SRE. We reviewed both academic work and popular plugins used by practitioners. We present aggregated answers to questions that are relevant to our study design. The survey and its answers can be found in Table **??** in the Appendix.

In total, 153 participants responded to our online survey. The survey respondents were students (43.1%), followed by employees (27.5%), academic researchers (17.0%), and freelancers (9.2%). Many participants were seasoned reverse engineers: 41.2% had more than three years of experience, while 40.5% had between one and three years. Their most used SRE framework was IDA Pro (81.0%), with Ghidra (73.0%) closely following.

LLM Use. Participants also reported how often they used LLMS during SRE: 34.0% responded sometimes, whereas 32.0% responded often or always. Additionally, 67.8% of participants reported that LLMs were occasionally beneficial, 25.2% considered them highly advantageous, while 7.0% found them unhelpful. The most used LLM was GPT (OpenAI, 85.6%), followed by Claude (Anthropic, 11.6%). To interact with their LLMs, participants reported mainly using decompilation as input (59%), followed by machine code (28%), and intermediate languages (13%).

LLM Features. Participants reported using LLMs for six identified features known from prior work:

(1) *Function Summarization*. Comments the function with a description or summary

based on its behavior or purpose [9, 57].

(2) *Function Identification*. Identifies well-known functions and algorithms. While testing this feature, we also noticed that if the LLM does not find a match, it tries to categorize the functions based on its potential role (e.g., cryptographic, network, file handling) w.r.t. similar well-known code patterns [74].

(3–4) *Function and Variable Renaming*. Renames the function or all of its variables with meaningful nomenclature to facilitate comprehension [9, 57, 74, 99, 12, 56, 103, 105].

(5) *Vulnerability Identification*. It identifies potential security vulnerabilities by detecting patterns and coding practices that may lead to security risks [74, 29].

(6) *Library Function Documentation*. It generates context-rich documentation of a library function so that the reverse engineer can understand its functionality without manually searching through reference materials [12].

The following percent of participants reported using these features in SRE: Function Summarization (63.7%), Function Identification (28.8%), Function Renaming (23.3%), Variable Renaming (28.1%), Vulnerability Identification (17.1%), and Library Function Documentation (2.1%). We note that participants wrote in the Library Function Documentation feature through an "Other" option in our survey. We use the results of these responses to design what LLM features will be available to participants in the study (Section 3.5).

Perceptions on LLMs. In a free-response section, participants also shared their experiences using LLMs for SRE tasks. The reported experiences are mixed, acknowledging both the benefits and limitations of existing LLM-SRE tools. Participants report that LLMs excel at explaining the behaviors of decompiled code and improving code readability by renaming variables. They are also useful for understanding known algorithms and accel-

erating workflows by creating scripts for SRE frameworks. Regarding limitations, participants report that LLMs often produce incorrect or misleading responses, reducing trust and wasting time. LLMs often create generic or superficial explanations, and their effectiveness diminishes for large, obfuscated, or highly mathematical tasks.

3.5 Study Design

While our online survey in Section 3.4 provides a broad perspective of LLM use in SRE, we next seek to understand how LLMs augment the process at a more technical level. Thus, we design a behavior-focused experiment that captures granular details on SRE performances, with and without LLM assistance on modern tools. Such a contrast allows us to measure not only raw performance differences but also how the presence of LLM assistance changes analyst behavior, tool usage patterns, and solution strategies.

We designed and piloted our experiment in collaboration with a Subject-Matter Experts (SME) team comprising 13 expert reverse engineers: three study authors, six research group members, two industry professionals, and two academics with expertise in binary analysis and Capture The Flag (CTF) organization. Importantly, none of the SME team members participated in the study, ensuring unbiased data collection.

3.5.1 Study Overview

Motivated by recent work in human SRE [71], we design two CTF-style reverse engineering challenges and task participants with solving them. We design a fully browser-based study, where participants can access a virtual machine (VM), a virtual networked computer (VNC), that hosts the challenges and LLM tools, whose features were informed directly by the findings of our online survey (Section 3.4).

We provide each participant with a link to our online platform (Section 3.5.2). Upon visiting the site, the participant is given access to two challenges (Section 3.5.4), ordered

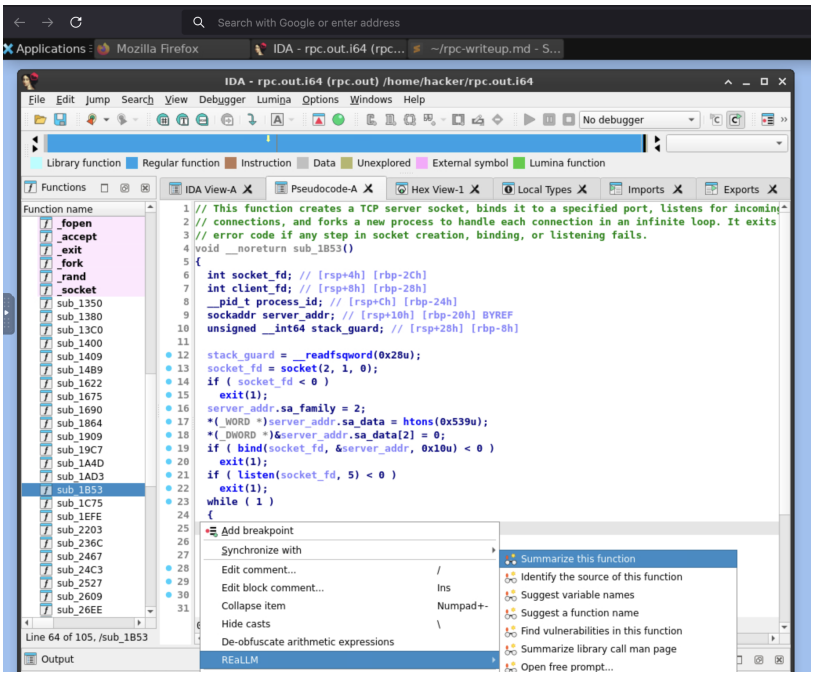

Figure 3.1: A screenshot of the online platform after a participant starts a challenge. A VNC is accessible in the browser that provides access to an instrumented decompiler (with an LLM plugin), browser, and text editor.

randomly. Upon clicking start on a challenge, a VM is launched, giving access to that binary, an instrumented version of IDA Pro (including the 6 LLM features derived from the pre-study, § 3.4), a text editor, and a web browser. On one challenge (random), the LLM features are disabled.

A participant reverses the challenge with the goal of *understanding* how to "read the flag." When a participant feels satisfied with their understanding, they end the study and complete the writeup. Upon finishing both challenges, they fill out a post-study survey. Accordingly, each participant in the study creates two writeups describing their solutions. We grade each writeup for understanding (§ 3.5.6) and collect the technical data, such as clicks, generated while they reversed.

3.5.2 Online Platform

A key aspect of our study design is making it accessible, since expert reverse engineers are difficult to find and schedule [35]. To accomplish this, we design our online platform to host our challenges and tools that can allow participants to complete the study asynchronously.

Our online platform builds atop the cybersecurity education work DOJO [80], which helps provide participants with a virtual desktop, a virtual networked computer (VNC), that hosts both the study challenges and our tools. A screenshot of what participants would see in this environment can be seen in Figure 3.1.

Upon visiting the site for the first time, a participant is shown our two challenges in a random order. Before starting the experiment, each participant could optionally view a walkthrough video introducing the IDA Pro interface and a basic toy program to familiarize themselves with the platform. We clearly state in each challenge's description that the participants must not find and report a flag, but *must provide a natural language description of the flag retrieval procedure*. This is the same approach that Mantovani et al. adopt [71] to avoid unintended solutions in their SRE challenges.

When participants click "Start" on a challenge, the recording for that VNC is started as well as all other instrumentation, and a VNC tab is opened. That VNC has both the challenge open in our tools and a text editor for creating a writeup. When a participant feels they have completed the challenge, they click "Finish" and the final time is saved along with their instrumented data and writeup.

3.5.3 SRE & LLM Tooling

When a VNC is opened, a participant is given access to an instrumented version of IDA Pro 9 with the respective challenge open in it. Our instrumentation collects most trackable

data in IDA, including functions visited, clicks, variable renames, etc. IDA is the only tool participants have access to, meaning they cannot use a debugger or run the challenge.

To give participants access to LLM features, we develop a decompiler plugin that implements the six features discovered in our online survey (§ 3.4). We implement each feature based on prior work [9, 57, 74, 99, 12, 56, 103, 105, 29]. In particular, we closely studied the methodology for creating prompts from Hu et al. [56], which created prompts to improve decompiled functions. We created our prompts by first describing the decompilation improvement task (summarization, renaming, etc.). Next, we described the output format of the task (JSON). We then provided a minimal example input with the correct output format. Finally, we provided the decompilation text of the function in question (triggered by participant use on a function). We validated this design choice by studying other related open-source prompts [57, 99, 9, 39, 74] and found that they had similar patterns.

Additionally, to account for any missed features that participants may use in real-world reversing, we integrate a chatbox into the decompiler where participants can interact freely with an LLM. Excluding the LLM chatbox, participants use the LLM features by right-clicking on an open function in IDA Pro and using the specified feature.

For all features, including the chatbox, participants could use any of the models reported in our online survey (§ 3.4): GPT-4o, GPT-4o-Mini, GPT-4-Turbo, Claude-3-5-Sonnet, and Gemini-Pro. When participants start the study, they are required to choose a model. If they are unsure which model to select, GPT-4o is used by default.

3.5.4 SRE Challenges Development

Here, we present our design heuristics for developing two challenges, RPC (Challenge #1) and Secrets (Challenge #2).

Representativeness. Challenges must represent real-world software. In collaboration with the SME team, we identify the most common software characteristics encountered dur-

Table 3.1: Software metrics of the SRE challenges.

Metric	RPC	Secrets
Functions	26	26
Lines of Code	427	461
McCabe Cyclomatic Complexity	29	31
Operands Count	957	1021
Distinct Operands	172	184
Operators Count	1933	2397
Distinct Operators	50	51
Halstead Volume	22526	26922
Halstead Difficulty	139	141
Halstead Effort	3.13×10^6	3.81×10^6

ing SRE and select a subset for manual static analysis. We exclude the ones that demand dynamic or automated analysis, e.g., self-modifying code. The list includes authentication/authorization, compression, encryption, hashing, data validation, encoding, file management, memory manipulation, networking, parsing, serialization, string operations, and sorting. We also include some known algorithms in each challenge because real-world software uses them. We note that this list excludes traditional program obfuscation, such as control flow flattening, which was outside the scope of our study. Our challenges also only contained advanced mathematics in hashing and encryption.

In our post-survey, we ask participants to rate their agreement with the statement "the two challenges are good representative examples of real-world software." Out of 48 respondents, 81% (n = 39) strongly or moderately agree whereas no one strongly disagreed.

With the help of the SME team and some novice volunteers, we estimate that both challenges could be solved within one hour for experts and two hours for novices. Our participants could complete both challenges at different times to reduce cognitive burden.

Equal levels of difficulty. The two challenges must act as control and treatment. So,

they must be of an equivalent level of difficulty, which means solving them should require similar efforts. However, perceived difficulty is extremely subjective. In addition to the supervision of the SME team, we elect to rely on the well-known software metrics, striving to ensure as much similarity as possible, as reported in Table 3.1. Particularly, the Halstead Difficulty measures the difficulty in understanding the program, for example, when doing a code review.

In our post-survey, we ask participants to rate their agreement with the statement: "The two CTF challenges were of similar difficulty." Out of 48 responses, 87% (n = 43) selected strongly or moderately agree, whereas only one strongly disagreed.

3.5.5 SRE Challenges

Challenge#1 – RPC. It mimics a remote procedure call (RPC) server. The server uses a custom protocol to communicate with the client, which involves encoding (Base64), compression (Run-Length Encoding - RLE), and encryption (Tiny Encryption Algorithm - TEA). In this protocol, after authentication, the client can manipulate a 32-byte buffer by sending specific bytes that correspond to different operations. The flag is stored in the `/flag` file, but only the admin user can load files into the buffer. Therefore, to solve this challenge, the player has to find the (hardcoded) admin user name and then spot a vulnerability: the decryption key of the admin password is easily guessable as it is randomized from the `time` library function. Finally, after authenticating as an admin, they must send the correct bytes with a sequence of operations that load the flag file into the buffer, which is eventually returned.

Challenge#2 – Secrets. It is an encrypted file manager with a custom shell (and commands) for managing files. The file manager supports multiple user accounts for creating encrypted files. The encryption algorithm is Advanced Encryption Standard (AES) in ECB mode. On user registration, a username and password are stored. The system will create

a folder with the same user name and derive the user's key using a custom key derivation function. Then, the key is hashed (DJB2), and its hash is stored in clear text in a file named `/<username>/.shadow`.The saved hash is from the key derived from the password. Since each file is encrypted, they are world-readable, implying that all other users can read everyone's hashes. In this system, the `admin` user has created an encrypted file named `flag` that contains the string to be retrieved. This challenge is solved by understanding that the function for showing the content of files is vulnerable to a path traversal attack. After creating a user account, the command `show ../admin/.shadow` will display the `admin`'s key hash. The custom key derivation function is then susceptible to brute-force attacks, as its output is contingent upon a mere two bytes and can be validated with the known DJB2 hash. In the event of a match, such a key decrypts the flag file.

Completion Criteria. Participants completed a challenge if they created a writeup and did not use outside tools other than IDA Pro and our LLM plugin. All writeups should contain text about solving the challenge. Writeups with only irrelevant text are not considered and would be an incomplete SRE session.

We note that this approach allows for participants who had incomplete or incorrect solutions to the challenges. We included all who made valid attempts on both and demonstrated some understanding of at least one. This variability makes direct comparisons difficult, as some participants spent less time and showed lower comprehension. However, it allows us to make a more holistic measurement of different SRE approaches and understanding levels. To account for this variability in solution completeness and participant understanding, we evaluated each challenge writeup using a structured scoring rubric, discussed next.

3.5.6 Challenge Writeup Evaluation

We scored each writeup from 0 to 8. Each challenge included 7 checkpoints, with a 8 point awarded for a fully correct solution. This accommodates solvers who demonstrate

holistic understanding without explicitly identifying all technical details (e.g., recognizing an algorithm's function without naming it).

Challenge#1 - RPC - Checkpoints. First, writeups were awarded one point for identifying the numerical operation to read the flag. To read the flag, they must use the hardcoded name and password "admin" and "secret," which is worth one point. The second bug in RPC is that randomness is seeded to time, which awards another point for mentioning it in their writeup. All data must be decrypted to talk to the server, and mentioning "encrypted" or "hashed" data also awards one point. Finally, a point is awarded for each of the three known algorithms (Base64, RLE, TEA) mentioned in the writeup.

Challenge#2 - Secrets - Checkpoints. Writeups are first awarded a point for mentioning that the program manages and creates files. An additional point is awarded if they mention that the files can be encrypted. A key part of the challenge is understanding that each password has a derived hash stored in a ".shadow" file, which awards another point. The hash is created from a key derived from the password, which is flawed. A point is awarded for mentioning that the derived key is flawed. To read an arbitrary user's hash, a user must use the path traversal in the "show" command. A point is awarded for mentioning "path traversal." Finally, a point is awarded for each of the two known algorithms (AES, DJB2) mentioned in the writeup. Note that Quicksort is not awarded a point since the algorithm is implemented in a function (ls) that is not required to solve the challenge.

Since the awarding of individual points could still be subjective, two researchers on the SME team scored the writeups independently. Across both challenges, a total of 96 write-ups were scored. Both researchers scored a writeup the same, on the same checkpoints, in 82 cases (85%). They differed by assigning one point on 12 cases (13%) and two points on 2 cases (2%). In total, both researchers aligned on 753 out of 768 point assignments (98%). With this in mind, we concluded that our grading criteria and point assignment were likely sufficient.

3.6 LLM Impacts on SRE

We ran our study, designed in Section 3.5, between 2024 and 2025. In total, 51 respondents from our online survey participated in our experiment. After reviewing their results, we eliminated 3 participants' data from our analysis due to submission quality: one left an empty writeup, another wrote only irrelevant text in a writeup, and one used dynamic analysis to solve their challenge. As such, all analysis is performed on the 48 valid submissions in our study. Cumulatively, these 48 participants reverse engineered for 109 hours across two challenges, creating 96 writeups, and interacting with LLMs 1517 times. This group included 24 self-reported experts and 24 self-reported novices. We report the summary of their performance in Table 3.2.

Therefore, the purpose of this section is to analyze these data to answer **RQ2:** *How does the inclusion of LLMs in the SRE process impact the performance of practitioners?*

We first focus on how LLMs affected the SRE process positively or negatively. Second, we analyze the strategies these participants took to get optimal performance from their LLMs. When comparing any two groups for independence or correlation, we use the methodology described in Section 3.3.3. When we obtain a statistically significant result, we report it with triple: p-value (p), effect size (es), and effect size interpretation. Otherwise, we just report the p-value.

Finally, when available, we integrate responses from our post-study survey to give a more holistic view of our results. Each response is anonymized to their skill level with *N* or *E* for novice and expert, respectively.

3.6.1 Study Assumptions

Before further analysis, we validate two assumptions made in our study design.

Assumption 1: *Self-reported SRE expertise is trustworthy.* To evaluate this assumption,

Table 3.2: Averaged performance metrics across all study participants and challenges, grouped by expertise.

Metric	Experts		Novices	
	RPC	**Secrets**	**RPC**	**Secrets**
Solve Times (m)	64.55	69.32	70.70	69.89
Time in Function (m)	2.39	2.56	2.62	2.61
Comments	2.00	1.96	1.38	0.83
LLM Interactions	19.96	10.42	19.17	13.67
Function Transitions	324.25	219.75	356.42	212.08
Clicks	512.04	361.17	636.12	417.21
Understanding Score	5.38	4.21	3.79	2.46

we first examined whether participants' self-reported years of SRE experience differed between the two groups. We found a statistically significant difference (p = 0.0001, es = 0.65, large effect size). On average, participants in the expert group reported six years of experience, compared to two years in the novice group. Although this comparison relies on self-reported measures as well, it provides evidence of internal consistency within participants' own assessments of their background.

We next evaluated whether the self-reported novice and expert groups differed on performance-based measures independent of self-report. Because we are interested in participants' underlying SRE skill, we compared their performance on the challenge completed without LLM assistance.

Experts statistically differed from novices in their understanding scores from their writeups (p = 0.001, es = 0.56, large effect size). On average, experts had an understanding score of 4.79, while novices had 3.12, both out of the maximum of 8. This indicates that the self-reported expert group often understood more than the novices across both challenges.

An *understanding rate* can be calculated by dividing a participant's understanding score by their solve time (minutes). Experts statistically differed from novices in their under-

standing rates (p = 0.001, es = 0.57, large effect size). Experts had an average rate of 0.08 points per minute, while novices had 0.06. This indicates that self-reported experts understood more in less time than novices, reproducing results found in previous work [71]. Although this data does not indicate if a self-reported expert is an expert relative to all reversers, it does suggest that they may be relative to our self-reported novices. Considering these observations, we assume self-reports of expertise levels are reflective of reality for our study.

Assumption 2: *Both challenges are of equal difficulty.* We validate this assumption by, again, comparing two groups when they are *not* utilizing an LLM. For this case, we compare participants across both expertise levels on the two challenges. There was no significant difference in understanding scores (p = 0.68) or rates (p = 0.47) on both challenges across all users, irrespective of expertise. Additionally, considering expertise, experts and novices had no significant difference in understanding scores or rates between the challenges ($p > 0.05$). This data indicates that challenges had similar rates of understanding and, therefore, difficulty.

3.6.2 Expertise Dependent LLM Improvements

SRE expertise plays a fundamental role in how fast and well reverse engineers understand a program [71]. SRE expertise also determined whether a participant greatly benefited from an LLM. In Figure 3.2, an overview of understanding rates can be seen across skill level and LLM use.

Finding 1. *Novices using LLMs exhibit expert levels of program understanding rates—regardless of whether experts use LLMs or not.* Comparing novice participants' performances across both challenges, we observed a statistical difference when they utilized an LLM (p = 0.03, es = 0.38, medium effect size). While utilizing an LLM, they had an average understanding rate of 0.07, compared to a 0.04 rate without (98.55% improve-

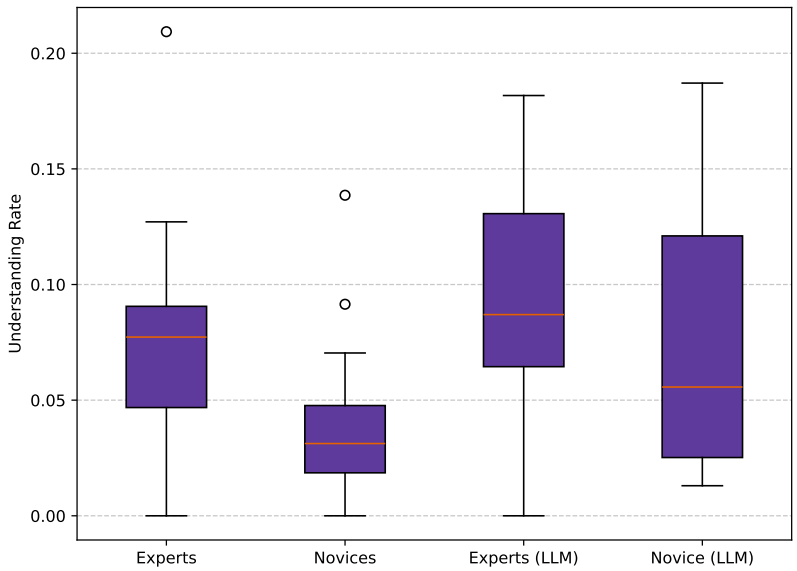

Figure 3.2: Understanding rates for participants with and without LLM assistance, grouped by SRE expertise.

ment). Additionally, when comparing novices with LLMs to experts (both with LLMs and without), we found no statistical difference in their understanding rates.

Various *novices* in our study also felt that they had improved holistically across challenges. As N15 noted, "I would not have understood the binary half as well without the LLM in that same time span." N13 remarked, "I was kinda mind blown by how the LLM analyzed some [complex] functions… and determined they perform base64 encoding and TEA decryption almost instantly. That really saved time during reversing by not having to focus on those parts and focus on more interesting parts to reverse engineer."

N14 summarized: "Using LLM made things much faster and easier with the risk of mistakes. However, these mistakes can be easily reversed. The time complexity spent on reversing might still be O(Time spent without LLM), but with LLM, the average time will be highly decreased." Meaning that in the worst case, fixing LLM mistakes could take as much time as doing the task manually from scratch (hence, same time complexity). However, this claim did not necessarily hold for all participants.

Finding 2. *LLM usage does not impact experts' program understanding rate.* Unlike

novices, when comparing experts' performances across both challenges, we did not see a statistically significant difference in their understanding rates with LLMs. However, experts had marginal gains in their average understanding rate with LLM usage. This data may indicate that SRE expertise outweighs the effects of LLMs. An expert (E5), articulated why this may be the case, stating that LLMs were "a hit or miss on usefulness, but low effort to read the summary and see if it seemed to be useful. Really high value when it got it right."

3.6.3 Augmented Artifact Recovery

During the SRE process, participants recovered different artifacts lost in compilation. Specifically, symbol names (both functions and variables), types (including custom structs), and comments. Prior work had identified that these artifacts (function names and comments) were recovered more frequently by experts [71]. We explore these findings by expanding the number of artifacts users can create and receive help on from an LLM. Except for structs, LLMs could also recover these artifacts on demand, as a practitioner would normally.

Finding 3. *Manual artifact recovery is positively correlated with program understanding.* We find that experts and novices did not significantly differ in the number of artifacts they manually recovered (p = 0.33). On average, experts recovered 54 artifacts, while novices recovered 41. We speculate that this difference in results from prior work may be caused by the expanded set of artifacts we support in our study, such as structs and stack variables.

However, we found that participants who manually recovered more artifacts, regardless of expertise, tended to have higher understanding scores. A higher manual artifact recovery rate was positively correlated with a higher understanding score (p = 0.01, large effect size). Manually recovered artifacts do not include artifacts directly recovered by an LLM. In fact, LLM-recovered artifacts did not show a similar correlation with manually recovered

artifacts.

Finding 4. *LLM artifact recovery is not correlated with improved program understanding.* We measured LLM recovered artifacts as those that an LLM directly created. This excludes artifacts that a human may have manually created after reading an LLM answer. These LLM changes were not correlated with improved understanding (p = 0.34). This finding may indicate that LLM recovered artifacts are not of the same quality as those created by humans. We note that this analysis ignores human changes resulting from LLM changes, which would increase understanding (explored in Section 3.7.2).

Finding 5. *LLM users recover more artifacts, including more false positives.* When combining the artifacts recovered by LLMs and humans, we found that the presence of an LLM led to more overall artifacts on a challenge. Both experts and novices significantly differed in their total recovered artifacts while using LLMs (p = 0.01, es = 0.52, large effect size, and p = 0.01, es = 0.43, medium effect size, respectively). On average, experts went from 54 artifacts to 84, while novices went from 41 to 67. Although these findings seem intuitive, they indicate that LLMs may not be replacing human efforts to recover artifacts. Considering that LLM recovered artifacts had no understanding correlation, our data may indicate LLMs create more artifacts that do not contribute, meanwhile increasing noise, which may be harmful.

An artifact type of particular interest was function names. Generally speaking, (good) function names summarize the purpose of a group of code. This abstraction allows SRE practitioners to understand what a function does quickly [59]. As such, creating a function name can often give insights about a participant's understanding of that function.

Finding 6. *LLMs recover function names of known algorithms with a higher accuracy than humans.* A *known algorithm function* implements algorithms whose designs and specifications are standardized and documented online, such as base64 decoding. Across both challenges, there was a total of 13 known algorithm functions.

We manually compared the renames from LLMs to those of humans on all 13 functions. A function rename was counted as a match if it was semantically equivalent. For example, with the ground truth being `handle_request`, a rename of `handle_connection` would be counted as semantically equivalent. LLMs showed a statistically significant difference in average rename accuracy for these functions when compared to humans (p = 0.4, medium effect size). On average, LLMs had an 85% accuracy, while humans 66%.

We also found that LLMs showed no significant difference in rename accuracy on all other functions when compared to humans (p = 0.82). Humans had an average accuracy of 65%, while LLMs had an average of 57% across 37 functions. Considering earlier LLM artifact correlations, we speculate the impacts of these renames may not significantly contribute to improved understanding.

3.6.4 Function Speed Differences

While solving both challenges, participants had to analyze various functions. We recorded both the total time participants spent in each function and the number of visits. We counted a visit as a participant looking at a function (in IDA Pro) for more than one second. On average, experts revisited a function 9.46 times and spent a total of 296.72 seconds in a function. Novices revisited a function 9.93 times and spent a total of 313.98 seconds in a function. Interestingly, we did not observe any correlation, suggesting a contribution from LLMs when viewing the sessions as a whole. Therefore, we focused on individual functions.

Finding 7. *Experts using LLMs spend less time on known algorithms and more time on custom ones.* At the binary level, experts showed non-significant performance when using an LLM. However, when comparing experts' total view times of functions across themselves, with and without LLM, they showed a significant difference ($p < 0.05$) in total view time on six functions. On four of these functions, LLM users were at least 248% faster

on average. All four functions implemented common algorithms: TEA, Heapsort, Base64 Decode, and RLE compression. On Base64 Decode, experts also had two fewer visits on average.

On the other two (out of six) functions, which implemented custom algorithms, LLM users experience a slowdown. They were a minimum of 314% slower on average, and had to revisit one of those functions, a request handler, up to 14 times more on average. Although these gains did save experts time, we conclude, from previous findings, that these did not play a pivotal role in understanding the program faster.

Finding 8. *Revisit frequency scales strongly with lines of code for everyone, and LLMs rarely change that pattern overall.* Looking at the binary as a whole, revisits are positively correlated ($p < 0.05$) with lines of code, in *every* case: for all participants, experts, and novices, and those with and without LLMs (six instances in total). While this is an expected result, it is surprising that Spearman's ρ we obtained is very similar in all the tested cases, i.e., $0.76 \leq \rho \leq 0.77$ (large effect size). We further investigated at function granularity, and in the majority of cases, both novices and experts did not show significant differences ($p \geq 0.05$) in their function visit times with and without LLMs. They did, however, show differences in function visit amounts on two functions. On the first, `load`, a function from RPC that contained 37 lines to read the flag, novices had 53% less revisits with LLM. On the second, `ls_cmd`, a function from Secrets that contained 107 lines to implement a Unix-style file lister, novices had 240% more revisits (with and without LLM). We note that the `ls_cmd` function was not essential to solve the Secrets challenge. Indeed, mentioning the function, or its purpose in the writeup, did not necessarily earn understanding points. This is a clear sign of how experts' skills have prevented them from wasting time on this function.

3.6.5 Misunderstandings

In some solution writeups, participants described program functionality that was incorrect or did not exist, which we classify as a *misunderstanding*. In 20 out of 96 writeups (20%), a user had at least one misunderstanding in their writeup. These misunderstanding cases were split evenly between 10 LLM users and 10 non-LLM users.

We analyzed these instances of misunderstandings and classified each occurrence into two distinct groups: *fabrications*, which were observed when a participant described a non-existent phenomenon, and *misclassifications*, which occurred when a description was incorrect. **Finding 9.** *LLM vulnerability hallucinations negatively impact subsequent human analysis.* We observed three fabrications only in the responses of participants who used an LLM. Specifically, all three participants reported a buffer overflow vulnerability in their writeup for the RPC challenge, which had been suggested by the LLM when asked to identify potential vulnerabilities. All three participants received these suggestions in different functions. On these functions, they spent a minimum of 231% more time than average.

Indeed, asking an LLM to identify vulnerabilities frequently resulted in worse user performance. The "find vulnerability" feature was negatively correlated with understanding score ($p = 0.01$, $es = -0.35$, large effect size).

Additionally, the LLM feature to identify vulnerabilities drew the most skepticism ($n = 5$) of all LLM features in our study. It was widely seen as unreliable, prone to false positives, and lacking actionable specificity. E22 wrote, "It said there was a UAF, but it's not." N15 commented, "After using it, I felt at that point I was better off searching for the vulnerability myself." E10 similarly shared: "ChatGPT is a game changer for reversing [...] really good for code understanding, not good for finding complex vulnerabilities."

3.7 LLM Strategies and Factors

In Section 3.6, we answered RQ2 showing how LLMs affect both novices and experts alike during the SRE process. A part of that analysis showed that novices approach a similar understanding rate to experts while using an LLM. In this section, we look at this phenomenon from a different angle by answering **RQ3:** *How do practitioners interact with LLMs, and what factors influence their interactions?*

Using a methodology inspired by previous work [71], we select the top ten and bottom ten participants by their understanding rate, on their LLM-enabled challenge, and we examine strategies and factors that contributed to improved or worsened performances. The top ten performers included five novices and five experts, whereas the bottom ten comprised nine novices and one expert.

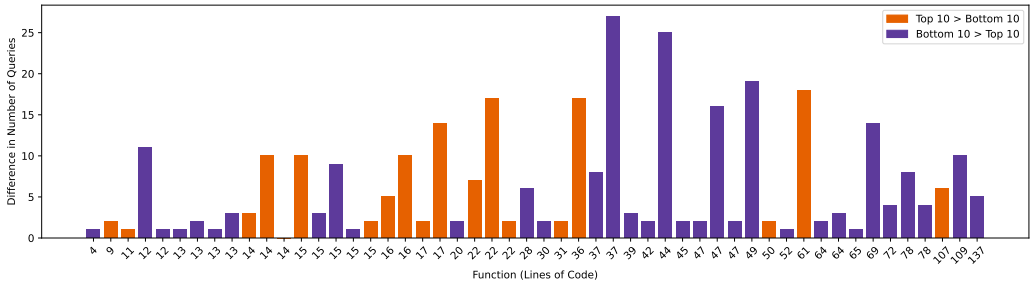

Figure 3.3: The difference in query amount by the top 10 and bottom 10 LLM users, sorted by function lines of code.

3.7.1 LLM Expertise

Before joining our study, all participants reported both their prior experience in SRE and LLM-assisted SRE. For LLM-assisted SRE experience, participants reported using LLMs rarely, sometimes, often, or always for SRE. We considered those who reported often or always to be experienced LLM users and all others to be inexperienced. On average, a participant reported sometimes using an LLM, indicating they are inexperienced LLM users.

Finding 10. *Prior experience in using LLMs does not make them more useful for SRE.* When comparing the top and bottom performers, we found no significant difference between their reported LLM-SRE expertise ($p = 0.78$). When expanding this to the entire dataset of participants, we found no correlation between LLM experience and understanding rate or solve time ($p = 0.25$, $p = 0.41$). Both comparisons suggest that having prior experience in using LLMs may not increase their usefulness significantly over that of new LLM users. This may also suggest that LLMs are intuitive enough to use, that new users derive the same benefits from them as experienced ones.

Finding 11. *Experienced LLM users are more cautious about using LLMs.* Although experienced LLM users were not significantly better at SRE with LLMs, they did show differences in how much they relied on them. We found that LLM expertise (higher being more experienced) was negatively correlated with LLM usage amount ($p = 0.04$, $es = -0.3$, large effect size). On average, an experienced LLM user queried an LLM 20 times during a challenge, while an inexperienced user queried it 35 times. We note that this difference did not correlate with improved understanding, suggesting that this caution may have been inconsequential.

Combining these two findings, we conclude that LLMs may not require special training for usage in SRE and can be used effectively by novices. However, it may be harder to convince prior LLM users to keep using them widely, since distrust may be contributing to their caution, as explored in Section 3.6.5.

3.7.2 LLM Features

In our study, participants had access to the six LLM features (presented in Section 3.4) and a free-prompt LLM chat. Across all participants, the most frequently used task was largely `Func Summary` (625), followed by `Func Rename` (328), `Var Rename` (287), `Lib Func Docs` (84), `Func Identify` (84), `LLM Chat` (75), and `Func Vulns`

(34).

Finding 12. *Understanding does not hinge on the quantity of LLM queries.* At a high level, no feature was utilized more across the top and bottom performers. When comparing the amounts of queries both for each feature and also all together, we found no significant difference between the two groups ($p > 0.05$). On average, top performers queried 29 times, while the bottom performers queried 37 times. However, these two groups still performed significantly different (p = 0.01, es = 1.0, large effect), with a 0.17 and 0.02 understanding rate on average respectively. The data show that sheer volume of LLM queries has no predictive power: the decisive factor is the placement of queries, i.e., knowing which function to ask about and when to ask, rather than how often any given feature is invoked.

Finding 13. *Asking LLMs clarifying questions is still an important LLM feature.* The `LLM Chat` feature is different from other features because it allows participants to freely interact with the LLM, where users can provide arbitrary context. We analyzed participants' use of the `LLM Chat`, which was used a total of 75 times, and categorized the users' interactions by their purpose with respect to the six features. However, the largest share of queries (40.6%) was context-related, and we were unable to associate them with a feature (e.g., hash-breaking strategies and explanations of regular expressions). This indicates a firm reliance on the LLM for problem-solving support and help with articulating technical concepts. Then, the most common category/feature was `Lib Func Docs`, accounting for 25.0% of the queries, suggesting that participants frequently relied on the LLM as a quick-reference tool for detailed questions about specific functions. This was followed by `Func Summary` (15.6%), `Func Identify` (12.5%), and `Func Vulns` (3.1%).

During reversing, participants switched between LLM actions (queries) and human actions (artifact recovery). Some actions used in succession led to increased understanding rates. We investigated these action sequences in pairs. The heatmap in Figure **??** visualizes these pairs, focusing on which led to human actions.

Finding 14. *LLM Func Summary and Var Rename lead to better program understanding.* Across all participants, three LLM features were correlated with an increase in understanding rate when accompanied by a human change: (`Func Summary` → function rename), (`Func Summary` → function retyping), and (`Var Rename` → function rename). They had the following statistical results: (p = 0.01, es = 0.40, large effect), (p = 0.02, es = 0.32, large effect), and (p = 0.01, es = 0.42, large effect), respectively. Notably, an LLM summarization followed by a manual function rename had the highest occurrence of 154, which dwarfed the next largest of just 19 with an LLM variable and a function rename.

E1 appreciated "the ability of LLM to effortlessly summarize the function, renaming local variable names, and renaming function name." N13 found Var Rename feature useful to "filter out decompilation statements." Similarly E7 commented, "Variable names were a lot more useful than I expected; a quick way to increase readability." According to N21, "summarize function combined with rename variables is a GOATED combo that made everything easier."

Of all action tuples involving LLMs, only these three tuples had positive effect on understanding rates. The common occurrence of all of them is the ending human action. This finding aligns with earlier findings in Section 3.6.3 that manually recovered artifacts were correlated with improved understanding. From this, we conclude that LLM responses which led to human actions were likely the most helpful towards increasing understanding.

Finding 15. *Experts overestimate the usefulness of some LLM features.* In the post-study survey, participants ranked their perceived usefulness of features. We asked for both an explicit score (on a five-point Likert scale) and a free response if they felt anything was notable. All reported post-study scores can be found in Figure **??** in the Appendix.

In only two cases, Func Summaryand Func Vulns, user opinions aligned strongly with our observed results: summarization was useful and vulnerability identification was harmful. After the Func Summary feature (n = 24, § 3.6.2), Lib Func Docs (n = 8), Var Re-

name (n = 8), and Func Rename (n = 5) features were found helpful by our participants for improving code readability in the free response.

E11 mentioned "the [document] library call man page [feature] was useful in speed[ing] up the reversing process." Novices had contrasting view about Lib Func Docs feature. N12 appreciated the feature :"the ability to summarize functions and write man page descriptions directly in the decompiled view as comments was an excellent feature." N19, on the contrary said, "summarize library call man page injected big blocks of text as comments in the decompiled view which made it hard to read, preferred the color highlighted man pages on the web instead." We note that across all experts, there was no notable change in performance–as we have previously demonstrated, yet, many quotes and Likert-scale responses from experts contradict our observed results. Additionally, features such as manual page recovery played a lesser role than many participants thought.

3.7.3 Query Frequency

Although the best and worst performers had similar overall LLM usage, they differed when analyzed on a per-function basis. One specific aspect was the number of times users were prompted on a single function.

Finding 16. *LLMs have degrading benefits when utilized repeatedly on individual functions.* Top performers had fewer repeated query uses across all functions than bottom performers. Their average query use per function significantly differed (p = 0.0004, es = -0.94000, large effect size), with the top users using 1.71 queries and the bottom users using 3.43. This data may extend the findings in Section 3.7.3, furthering that less query use on individual functions may allow better understanding rates among participants.

Some participants reflected this fear of over-querying and over-reliance on LLMs: as E8 admitted, "I was prone to try to go faster and be less careful, relying a bit too much on the LLM support... sometimes it caused me to miss important details." This issue was

particularly found concerning for less experienced users, as cautioned by E9: "I wouldn't want a novice using it, as I'm afraid they would reverse a lot of programs without actually understanding what's going on."

The most frequently queried functions were `dispatch` and `decompress` from the RPC challenge, each queried 22 times. Specifically, `dispatch` handles the user's operation command, while `decompress` performs RLE decoding. Similarly, `base64_decode` and `tea_decrypt`, also from RPC, were queried 16 and 18 times, respectively. These functions all play a central role in the challenge logic, and their high query count may reflect areas of conceptual complexity or critical decision points for participants.

Finding 17. *LLMs perform worse on larger functions (greater than 40 lines of code).* The size of functions, which often relates to complexity [51], was also a factor in query reuse. For both challenges, the median, mean, and max lines of code for functions were 31, 38, and 137, respectively. The top performers significantly differed from the bottom performers in their use of LLMs on functions larger than the median size ($p = 0.0003$, $es = -0.57988$, large effect size). In Figure 3.3, the *difference* in query amount can be seen on each function across both challenges, sorted by lines of code. Interestingly, the top performers used LLMs *less* on larger functions than the bottom performers, with an average of 4.73 vs 9.69.

3.7.4 Query Timing

One factor of query use was the time at which a participant decided to utilize an LLM while reverse engineering a function. To better quantify the time spent reversing on a function, we also tracked the number of visits (leaving a function and then re-entering it) participants had on functions. We observed that many top participants used all their LLM queries on their first visit to a function. To further investigate, we measured the number of visits each participant made to a function before using their last LLM query, and we

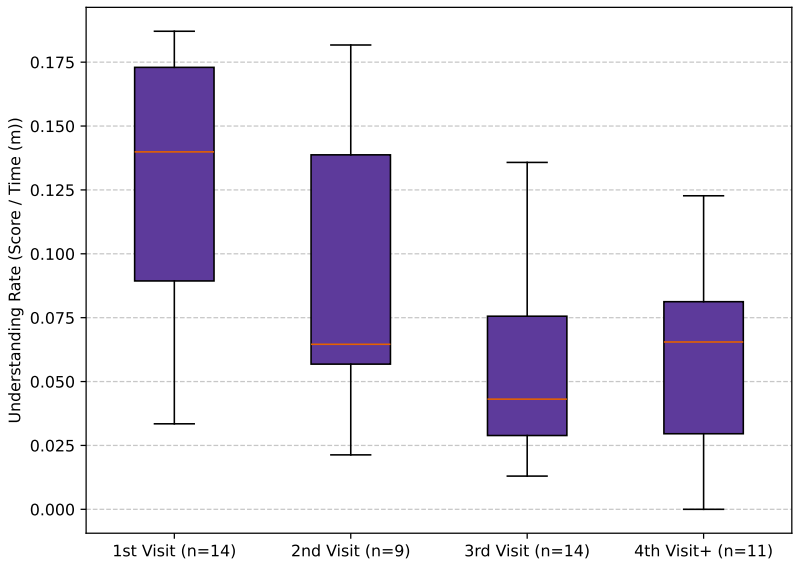

Figure 3.4: Understanding rates for participants that complete all queries on their first function visit vs on multiple visits.

categorized the strategy accordingly. For example, if a participant completed all of their LLM queries in three visits to a function, they would implement a three-visit strategy for most functions. The aggregated understanding rates of each visit strategy can also be seen in the Figure 3.4.

Finding 18. *LLMs provide greater benefit when utilized at the beginning of function understanding, not the end.* Out of the top ten performers of all LLM users, 9 are first-visit strategy users (out of a total of 11 first-visit strategy users). Moreover, first-visit strategy users significantly differed in their understanding rates from all other participants (p = 0.01, es = 0.45, medium effect size). On average, they had an understanding rate of 0.13, while all users had 0.08. This data indicates that LLM use at first glance of a function may produce better overall results.

Users who adopted the first-visit strategy primarily relied on the `Func Summary` prompt, which accounted for 41.20% of their queries. This was followed by `Var Rename` (18.92%) and `Func Rename` (21.62%). In contrast, the use of `LLM Chat` was relatively limited (4.94%), with `Lib Func Docs` (5.54%), `Func Vulns` (2.24%), and `Func`

`Identify` (5.54%) used even less frequently.

3.7.5 Other Strategies

Participants also described other LLM strategies that they felt had a significant effect on their SRE process. E7 noted, "When I renamed a few things first, I think the LLM suggestions got better." E12 noted that they observed improved results when they added manual context, such as function comments, saying 'I often had to add a bit of context in a hack-y way." However, we found no correlation between any human-action-to-LLM chain and understanding score, suggesting that this may be a perception rather than a reality.

E22 shared their strategy of working from the inside out, naming variables and functions in the deepest functions first, then moving outward, also known as a depth-first search (DFS) strategy, and added that "naming them from inner function to its caller provided better results." Previous work has shown DFS to be an effective strategy for humans in SRE [71], and we found indications that this may also be true for LLM use. When comparing visit time on individual functions, we found that functions queried in a DFS strategy versus a linear strategy had a statistical difference (p = 0.001, es = -0.24702, small effect size). The averages for each were 25.82 seconds and 46.83 seconds, respectively, though a small effect indicates that there may not be a substantial difference in results.

3.8 Key Takeaways

We conducted the first empirical study comparing humans, both experts and novices, with and without LLM support during software reverse engineering. In this section, we synthesize the major themes emerging from our findings and discuss their broader implications.

I) LLMs primarily shape the first moments of understanding, not deeper refinements. Our findings indicate that the SRE process is often disproportionately influenced by the first

encounter with a function. Early impressions frequently determine the pace and quality of subsequent analysis, affecting both overall solve time and understanding rate. LLMs are well-suited to this stage, as they enable practitioners to obtain a first-pass overview at minimal cognitive costs. This effect is most evident among novices, whose understanding rates approach those of experts when assisted by LLMs (Findings 1, and 18).

However, this advantage does not extend to deeper or iterative understanding. When practitioners revisit functions or attempt to refine mental models across functions, LLM assistance yields diminishing returns and can even hinder progress (Finding 16). These results suggest that current LLMs act primarily as summarization tools rather than collaborators capable of supporting sustained reasoning across functions or binaries. As a result, LLMs accelerate early semantic grounding but do not meaningfully assist with the deeper analytical processes characteristic of expert-level SRE (Findings 2, and 7).

II) The current interaction model of LLMS in SRE does not support expert knowledge refinement. Despite increasing adoption, the prevailing interaction model for LLM-assisted SRE does not effectively support experts. Experts exhibited little improvement in understanding rate or overall performance when using LLMs, suggesting that the integration model itself is insufficient (Finding 2). Experts selectively offload routine pattern recognition tasks, such as identifying known algorithms, while reallocating effort toward bespoke or complex logic. This results in effort redistribution rather than net acceleration (Finding 7).

These observations highlight a limitation of current LLM integrations, which primarily treat LLMs as first-pass summarizers rather than tools for iterative refinement or collaborative hypothesis development. Without advances that enable meaningful knowledge refinement, LLMs are likely to remain assistants for novices rather than true teammates for experts.

III) Even rare hallucinations can severely derail SRE workflows. Although hallucina-

tions occurred infrequently across thousands of LLM interactions, their impact was disproportionately harmful. In particular, hallucinated vulnerability reports significantly disrupted participants, often leading analysts to pursue nonexistent flaws for extended periods (Finding 9). This behavior is especially concerning in SRE, where practitioners frequently explore competing hypotheses and cannot easily invalidate theories without substantial investigative effort.

IV) Semantic recovery often depends on the act of naming, not merely the presence of names. Artifact recovery, including function names, variable names, types, and comments, plays a central role in SRE. While LLMs substantially increase the total number of recovered artifacts, these automatically generated artifacts are not correlated with improved program understanding (Findings 4, and 5). In contrast, artifacts created manually by practitioners show a strong positive relationship with understanding (Finding 3).

We hypothesize that this discrepancy arises because artifact creation is itself an understanding process. The act of naming requires the analyst to commit to a hypothesis about a function's role, even when imperfect. When LLMs generate artifacts automatically, this process is partially bypassed, potentially reducing opportunities for deeper comprehension. While LLM-generated names are effective for identifying known algorithms (Finding 6), reliance on automatically generated artifacts in more complex contexts may hinder understanding.

V) Only certain forms of cognition can be effectively offloaded to LLMs. Our findings reveal clear limits on which aspects of SRE can be effectively offloaded to LLMs. Tasks that admit a concise, single-purpose abstraction, such as summarizing well-scoped functions or identifying standard algorithms, are well suited to LLM assistance and consistently beneficial (Findings 6, and 18). In contrast, functions that serve multiple roles, have been heavily optimized, or incorporate intertwined logic, remain challenging for LLMs to characterize accurately (Finding 17).

These results suggest that LLMs are most effective for compressible cognition, where behavior can be meaningfully captured in a single abstract description. For more complex code, successful SRE still requires iterative human analysis, potentially complemented by selective and early LLM use.

3.9 Related Work

Human Cognitive Processes during SRE. Mantovani et al. [71] investigated how human experts approach and solve SRE tasks. Although their study primarily focused on analyzing cognitive strategies, it provided a foundational reference for the design of our user study. Additional related work on characterizing mistakes in the SRE process [43] and on the automated analysis of instrumented disassembler SRE data via screen recordings [112] further informed our effort to understand human behavior in SRE.

In contrast, our work differs significantly in scope and methodology: we introduce more realistic and semantically rich challenges with higher code complexity, and our participants interact with a broader set of tools – most notably, decompilation and LLM integration – which fundamentally alter the nature of the SRE process and the cognitive dynamics involved.

Other researchers [6] evaluated whether the adoption of external machine learning-based tools can improve human performance in discriminating a malicious program from a benign one. The authors find that machine learning algorithms can be a valuable tool for malware classification but should not be used to replace human analysts entirely. While Aonzo et al. [6] compare the independent decision-making processes of humans and machines, we delve into how LLMs can assist human analysts throughout the RE process.

LLMs for SRE. Since the recent advent of LLMs, we have seen an increasing evaluation of the capabilities of LLM-based tools, some of which have also considered the SRE scenario. Concerning end-to-end SRE, Pearce et al. [87] explored the ability of LLMs to

identify program purposes, capabilities, and variables from decompiled code. Their findings indicate that, while LLMs show promise, their performance is insufficient for reliable zero-shot reverse engineering. Their work establishes that LLMs can perform some initial SRE tasks, but they do not study how their results may be used or perceived by human SRE practitioners. Our work explores this interaction and the more human-driven use of LLMs in SRE.

Amodei et al. [105] propose a novel technique to recover variable names from binaries, leveraging the strengths of both generative models and program analysis. They evaluated their technique on a dataset of 940k binary functions and showed that it outperforms state-of-the-art techniques. The recent paper by Peiwei et al. [56] directly addresses the synergy between decompilers and LLMs for SRE. Their focus on optimizing decompiler output with LLMs aligns with our goal of improving code readability during the SRE process. Similarly, Shang et al. [92] evaluated LLM's ability to understand stripped binary code, focusing on tasks such as function name recovery and code summarization. Their study revealed the challenges posed by complex and obfuscated code, emphasizing the need for further research to improve LLM performance in this domain. Rukmono et al. [91] introduce an approach to summarizing software systems at higher levels of abstraction. They combine the capabilities of LLMs with static code analysis to generate summaries of software components.

LLMs are also used to find vulnerabilities in code [94]. Researchers have mostly focused on C/C++, Java, and Solidity, in particular on memory-related, framework-specific, and smart contract vulnerabilities, respectively. However, most attempts target function-level and file-level vulnerabilities, while there is a lack of repository-level datasets, thus limiting the adoption in real-world scenarios.

Our work builds upon the existing state of the art and offers a deeper understanding of the practical applications of LLMs during the whole SRE process.

User-Study on LLMs. Researchers have explored how users interact with LLM-based solutions for software development and analysis. LLMs have been used to improve developer productivity. GILT [78] introduced a prototype tool that integrates information retrieval with LLMs to improve information search during software development. The authors demonstrated the effectiveness of the tool in assisting developers with various tasks. Nguyen et al. [82] investigated how students interact with LLMs for code generation, highlighting challenges in prompt engineering and the misalignment between human and machine understanding. Furthermore, Zamfirescu-Pereir et al. [110] explored the difficulties non-AI experts encounter when designing prompts.

Previous user studies have focused on measuring user interaction with LLM-enhanced tools at individual stages of the SRE process or software development. This work focuses on the entire SRE process workflow.

3.10 Limitations

While designing our study, we had to make several choices to balance the feasibility of the study with the amount of data we could collect. These choices may have introduced bias into our results.

Restricted Tooling. In the SRE sessions of our study, we limited all participants' analyses of challenges to static analysis. In real-world SRE, practitioners often have access to dynamic analysis tools, such as debuggers, which enable them to confirm assumptions through direct observation. They may also have access to other, more advanced tools, like symbolic execution frameworks. Since we did not allow participants to use these types of tools, they may have solved challenges in alternative ways to their normal workflow. This limitation can contribute to longer solve times and understanding rates.

Additionally, we did not survey our participants to determine if they were satisfied with a static-only approach to reverse engineering. We speculate that many would prefer more

tools when reverse-engineering. However, to limit confounding variables in our study, we limited what tools participants could use, similar to prior work [71].

Challenge Representativeness. We designed two CTF-style challenges for our study that approximate tasks SRE practitioners may encounter in real-world programs. Although the majority of participants reported that they are representative of real programs (Section 3.5), our challenges can still lack characteristics of real-world software. Our challenges contain no obfuscation, have limited complex mathematics, and are smaller than modern statically compiled programs.

Self-Reported Expertise. To determine SRE expertise, we relied on self-reporting and partially confirmed results by statistically testing the difference between the two groups. However, the difference between these two groups may not necessarily indicate that one is a novice and the other an expert. Similar to previous works [71], this could have led to the incorrect classification of experts and novices in SRE. Future work should explore more effective experiments for benchmarking and determining expertise levels in SRE.

LLM Prompts. As with any research applying LLMs, the style and wording of prompts may influence the results. During the SRE sessions, participants primarily relied on prompts written by the research team. These prompts were adapted from previous research, but they pose the risk of being more or less effective than the prompts practitioners currently use. As such, some scenarios of LLM use in our study may yield better or worse performance, which could bias the results.

3.11 Conclusion

Our study takes the first in-depth empirical dive into how LLMs impact SRE and demonstrates that they *augment* analysts rather than *replace* them. The benefits are clear: LLMs support SRE practitioners in many tasks and can even close the gap in some aspects between experts and novices. However, our study shows that many areas of LLM inte-

gration still require research and development to be helpful for the most challenging SRE tasks. We present our findings, hopeful that the SRE community will utilize our work to make LLMs an even more helpful tool for security. All in all, LLMs are neither oracles nor impostors, but mirrors held to human insight: their reflections sharpen only in the steady gaze of critical thought and domain expertise.

3.11.1 Ethics Considerations

Our study constitutes human subject research (HSR), so we follow our institution's HSR guidelines and received approval from the Institutional Review Board (IRB). Our study is approved under "Continuing Review," which requires yearly re-approval to ensure continued ethical and safety compliance until all research activities (including reporting of findings) are complete.

In accordance with our IRB's requirements and our own ethical considerations of the proper handling of our participants, all of them were fully informed of the study purpose, data handling procedures, and their right to withdraw at any time without penalty. They provided explicit consent for recording and data storage, and all recordings were stored on access-controlled servers in accordance with best security practices. No sensitive or personally identifying information was collected beyond contact details necessary for compensation.

The study tasks were designed to pose minimal psychological or professional risk, with no deception or stress-inducing elements. Participants were not required to complete the two challenges in sequence, as they could take a break of any duration between them. Finally, all results were analyzed and reported in aggregate form to prevent any possibility of reidentification. Even when participant perceptions (qualitative insights from the open responses) are reported about their experiences in the experiment, we use random participant identifiers.

Chapter 4

STUDYING REVERSE ENGINEERING APPLICATIONS

4.1 Introduction

Software systems are becoming increasingly complex, while the number of vulnerabilities within them continues to grow. Despite this expanding attack surface, the population of developers and security experts capable of fixing these issues has not kept pace. In particular, modern society depends heavily on a small set of open-source maintainers who are often overworked and under-resourced, yet responsible for critical infrastructure [34]. When maintainers neglect these systems, the consequences are severe, as evidenced by incidents such as the XZ backdoor [83] and large-scale software failures impacting millions of systems.

As a result, security researchers have increasingly taken on the responsibility of identifying vulnerabilities in widely used software, as well as developing and deploying fixes. This task is fundamentally challenging. Analysts frequently work with unfamiliar codebases, lacking documentation, context, or original developer intent. The growing prevalence of code generated or assisted by large language models amplifies this challenge, as even the original authors often do not fully understand the resulting implementations.

In practice, this process closely resembles software reverse engineering: analysts must construct an understanding of a system from incomplete information and make high-stakes decisions based on that understanding. Rather than modifying well-understood code, they must infer behavior, intent, and constraints before safely applying changes. This requirement, to understand before acting, forms the core of reliable vulnerability repair.

Recent advances in large language models (LLMs) suggest a promising path forward.

LLMs have shown an impressive ability to reason over incomplete and ambiguous code, leading to progress in vulnerability discovery [48] and automated patch generation [60]. However, translating this capability into reliable program repair remains difficult. Existing approaches often rely on one-shot synthesis or open-ended agentic behavior [109, 113]. These methods are hard to control: analysts cannot readily inspect their decisions, their execution paths vary across runs, and their outputs often reflect weakly justified assumptions about program behavior. In the context of patching, such instability is especially problematic. A seemingly plausible fix often resolves one failure while violating intended functionality elsewhere.

We therefore argue that effective LLM-based patching systems should be structured around control as well as capability. Instead of allowing unconstrained generation, they should mirror the staged reasoning processes that human analysts use when repairing unfamiliar code. Concretely, this means making patching workflows observable, so analysts can examine intermediate results. Workflows should be more deterministic, so outcomes remain reproducible under controlled settings. Finally, workflows should be grounded, so each repair step draws on concrete program evidence rather than unsupported inference. A system designed this way produces fixes whose reasoning an analyst can follow and whose correctness an analyst can meaningfully assess.

In this chapter, we present PatcherY, a system for LLM-driven vulnerability repair designed around the intuition and workflow of software reverse engineers. Rather than directly generating patches, PatcherY decomposes the repair process into stages, including fault localization, behavioral understanding, and functionality verification, combining traditional program analyses with LLM reasoning to iteratively construct and validate fixes. This design enables PatcherY to produce patches that are effective, interpretable, and grounded in observable behavior.

We evaluate PatcherY as part of a large-scale autonomous cyber reasoning system in the

DARPA AI Cyber Challenge (AIxCC), where competing systems must discover and repair vulnerabilities in real-world open-source software. In this setting, PatcherY demonstrates its ability to scale to complex, previously unseen codebases while maintaining high patch correctness, highlighting the importance of structured, trustworthy design in LLM-driven security systems.

This chapter builds on the central theme of this dissertation by studying a concrete application of reverse engineering. While prior chapters focus on individual reverse engineering techniques and the processes by which analysts compose them, PatcherY examines how application demands, such as the requirements of real-world vulnerability repair, validate and inform the design of reverse engineering methods. In doing so, PatcherY illustrates why studying applications is essential to a science of reverse engineering: applications reveal which aspects of understanding matter most for effective intervention and which do not.

4.2 Background

4.2.1 Automatic Program Repair

Automatic Program Repair (APR) seeks to generate patches that correct program faults while preserving intended functionality. A large body of prior work follows a generate-and-validate paradigm, in which the system synthesizes candidate patches and evaluates them against a validation oracle, typically a test suite. Early approaches demonstrated that search-based techniques successfully identify repairs by exploring program transformations guided by test outcomes [65].

Despite these advances, a fundamental limitation remains: these systems approximate correctness using test suites, which are often incomplete. As a result, many generated patches satisfy all available tests while still violating the intended semantics of the program.

Prior studies have shown that such plausible but incorrect patches are common in generate-and-validate systems [88]. This limitation restricts the reliability of APR techniques in practice, particularly in settings where tests cannot fully capture correctness.

Recent work has explored the use of large language models (LLMs) for program repair. These approaches generate patches directly by reasoning over code, rather than searching over predefined transformations. Empirical evaluations on real-world datasets indicate that LLMs resolve a subset of bugs in existing codebases, suggesting their potential for practical use [60]. In contrast to traditional APR techniques, these approaches are not limited by a fixed set of transformations and operate over broader contexts, including incomplete or partially understood code.

This flexibility introduces new challenges. Probabilistic inference drives patch generation rather than a constrained search process, making system behavior more difficult to predict and reproduce. Generated patches vary across executions, and these systems often do not explicitly represent the reasoning process leading to a patch.

More recent approaches incorporate iterative or agent-based workflows, allowing models to explore codebases, invoke tools, and refine candidate patches over multiple steps [109, 113]. While these approaches expand the range of addressable problems, they further increase the complexity of system behavior. Execution paths depend on sequences of model decisions, and intermediate states are often difficult to inspect or reproduce. This makes it challenging to reason about the correctness and stability of the resulting patches.

4.2.2 Patch Verification

Patch verification aims to determine whether a generated patch is correct, both in terms of resolving the target fault and preserving intended program behavior. Most APR systems perform verification using test suites. This approach is efficient and widely applicable, but it inherits the limitations of the test oracle. Incomplete test coverage allows incorrect

patches to pass validation [88].

More precise verification techniques leverage program analysis, including symbolic execution, equivalence checking, and static analysis [81, 75]. These approaches detect classes of incorrect behavior that tests miss, but they often face scalability challenges on large or complex systems. As a result, APR systems typically combine them with lighter-weight validation methods rather than using them as a complete replacement.

Despite these efforts, verification remains a limiting factor. Lightweight validation techniques do not fully capture program intent, while stronger guarantees are difficult to obtain at scale.

4.2.3 DARPA AI Cyber Challenge (AIxCC)

The DARPA AI Cyber Challenge (AIxCC) was a two-year competition focused on automated vulnerability discovery and repair in real-world software [28]. The competition offered a $29.5 million prize pool and included teams from both academia and industry. It aimed to evaluate whether fully autonomous systems can identify and repair vulnerabilities in large, widely deployed open-source projects.

AIxCC consisted of a one-year semi-final round followed by a one-year final round. In the semi-finals, the organizers injected vulnerabilities into real-world open-source codebases through individual commits, with one vulnerability per commit. This structure bounded the scope of each task and allowed systems to focus analysis on a single change, implicitly constraining the search space for both vulnerability discovery and repair.

The final round removed this assumption. While part of the evaluation retained the commit-based format, the remaining tasks stripped all commit history and metadata. Systems had to operate on codebases without any information about where the organizers introduced vulnerabilities. This setting more closely matched real-world conditions, where analysts lack access to clean version histories and must reason over entire programs.

Teams submitted Cyber Reasoning Systems (CRSs) that autonomously identified vulnerabilities and generated patches. These systems operated with minimal or no human intervention. A central challenge was the lack of ground truth during execution. When a CRS produced a patch, there was no immediate feedback indicating whether the patch was correct or whether it introduced unintended behavior. This mirrored real deployment settings, where validation is often as difficult as detection. Regarding competition points, a *correct* patch was roughly equivalent to two discovered bugs, pushing teams heavily to validate patches before submission.

4.3 Motivation

Autonomous patching techniques have existed for decades, but they have often been constrained to simple cases [65]. These cases involve bugs with simple detection mechanisms (such as an overflow), inputs that crash relatively close to the crash site, and, notably, smaller programs. Real-world programs, by contrast, often contain faults that span hundreds of thousands of lines of code and involve indirect control flow or non-trivial logical errors. In the DARPA AI Cyber Challenge (AIxCC), many bugs went beyond simple memory corruption and required more advanced reasoning to patch.

In Listing 5, we examine a bug (labeled as CPV14) that AIxCC introduced into the open-source software Nginx, which is over 230,000 lines of code. AIxCC injected the bug into the function `ngx_http_script_regex_end_code`, which implements the final step of regex-based URI rewriting in Nginx's rewrite script engine. Upon detecting an overlong URI, the code correctly sets a terminal error state by redirecting execution to `ngx_http_script_exit`. The removal of the `return` on line six allows execution to proceed, causing subsequent operations to run under an inconsistent control-flow state (caused by the setting of `e->ip`). This produces an invalid transition through the script engine and ultimately triggers a global buffer overflow, crashing the server.

```
e->buf.len = e->pos - e->buf.data;
if (e->buf.len > 2000) {
    ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "the rewritten URI is too long");
    e->ip = ngx_http_script_exit;
    e->status = NGX_HTTP_INTERNAL_SERVER_ERROR;
-   return;
}

//...
if (r->uri.len == 0) {
    ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "the rewritten URI has a zero length");
    e->ip = ngx_http_script_exit;
    e->status = NGX_HTTP_INTERNAL_SERVER_ERROR;
    return;
}
//...

e->ip += sizeof(ngx_http_script_regex_end_code_t);
```

Listing 5: CPV14 from the DARPA AI Cyber Challenge Nginx source code. In this CPV, the AIxCC organizers removed the return in the first if-statement of the original validation code found in the `ngx_http_script_regex_end_code` function. This causes invalidated data to later be used in a memory write that assumes validated data, leading to a downstream global buffer overflow.

This bug exhibits several notable properties when considered in the context of a triggering input generated via a fuzzer. First, an indirect call triggers this bug, a construct that static analysis techniques have traditionally struggled to resolve [52]. When executed on the organizer-provided input, the crash occurs in the function `ngx_http_rewrite_handler`, which has no direct control flow to or from the intended patch site `ngx_http_script_regex_end_code`. A handler in the crash site calls the intended patch site indirectly. This indicates that approaches which patch only the crash site, or code that flows to it, will likely fail.

Second, *hundreds* of Nginx-written functions execute before this bug triggers. Excluding library calls to code outside of Nginx, there are 376 function calls prior to the crashing

function. Considering each of these 376 functions as a patch candidate is infeasible and would exceed the context window of most modern LLMs.

Third, the root cause of the bug is abstract rather than a memory error. When executing the triggering input, a sanitizer such as ASAN indicates that this bug manifests as a global buffer overflow. While technically true, this classification is deceptive: the true root cause is incorrect error handling, a type of fault that is notoriously difficult to classify. Traditional APR techniques cannot feasibly patch this bug, and LLMs often become confused about the root cause of the crash. A human easily sees that the second if-block uses the return and the first does not, but encoding this type of reasoning in an LLM is difficult.

Together, these problems motivate the need for automatic program repair methods that reason about complex situations where memory corruption is not the root cause. These approaches should be scalable, and, in the case of triaging and patching with LLMs, should be explainable and verifiable.

4.4 Design

Motivated by the complex patching scenarios in Section 4.3, we design PatcherY with the following properties:

- **LLM Centric**: Complex patches require a reasoning system that generates arbitrary code based on simple or incomplete descriptions. LLMs fit this role well for complex scenarios. We start with LLM capabilities, then build our system around them. Although the LLM plays a central role, we constrain its ability to use tools and restrict which functions it patches.

- **Stage Driven**: We design our system based on practical experience in reverse engineering and patching third-party software. The system mimics the stages we follow while patching, inspired by practice and previous work in studying reverse engineer-

ing [101]. In order, these phases are: *Root cause analysis*, *Candidate patch filtering*, *Code generation* (writing the patch), and *Patch validation*.

We describe each of these four phases and their sub-components below.

4.4.1 Overview

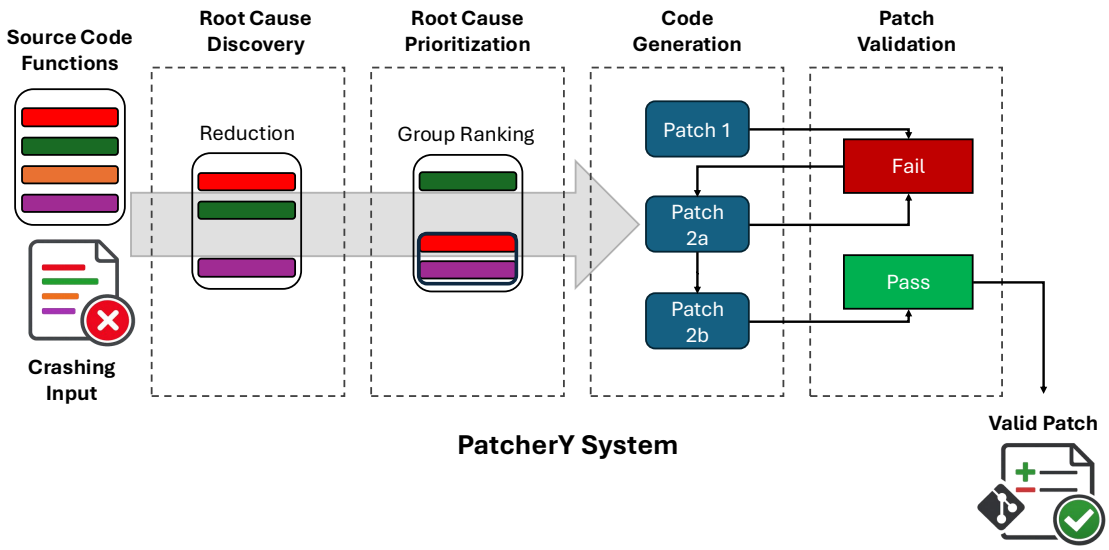

Figure 4.1: An overview of the four patching phases of the PatcherY system. The system requires the source code and crashing corpus as input and outputs a validated patch.

PatcherY operates as a staged system that incrementally refines its understanding of a vulnerability and constrains how it generates and accepts patches. Each stage consumes structured artifacts from earlier analysis (e.g., crash reports, stack traces, candidate functions) and produces increasingly concrete representations of a fix. This design reduces reliance on one-shot generation and instead decomposes patching into a sequence of verifiable steps. Figure 4.1 shows an overview of this system.

At a high level, PatcherY takes as input a verified proof-of-vulnerability (PoV), which includes the crashing input and the associated harness. From this input, it identifies can-

didate root causes, generates and filters candidate locations for patches, produces concrete code edits, and validates those edits against observed behavior.

4.4.2 Root Cause Analysis

Given a crashing input, PatcherY must first identify the location and cause of the vulnerability. This is non-trivial: the point at which a failure manifests is often not the location of the underlying fault [86]. For example, a memory violation often occurs at a dereference site, while the root cause originates from an earlier incorrect computation or transformation.

PatcherY combines structured program analysis with LLM-based reasoning to localize the root cause. The system first extracts execution artifacts from the crash, including stack traces and relevant program state. At the same time, PatcherY leverages prior work in statistical root cause analysis [86]. PatcherY selects the top 10 overlapping function candidates between these two sets as finalists and further prunes them via data flow analysis. Specifically, PatcherY discards any function that has no effect on the data associated with the crashing site. In practice, this reduces the search space from the entire program to a small number of functions for detailed inspection.

Rather than producing a patch directly, this stage outputs a ranked set of candidate locations and hypotheses describing the root cause.

4.4.3 Candidate Patch Filtering

After identifying candidate patch locations, PatcherY further ranks which location to patch first. This ranking is necessary because compiling projects after creating a candidate patch is slow. Even with only 10 candidate locations to patch, the total compilation time can exceed an hour.

PatcherY ranks each candidate using a weighted, per-function system based on the root

cause origins found in the prior phase. For example, a function identified as a root cause site from the stack trace is considered more trustworthy than a function found through statistical root cause analysis. Accordingly, the more root cause sources a function has, the more likely PatcherY is to select it as the first patch location. We tuned the weights for each root cause type over time through observed performance on targets from the ARVO dataset [76].

Since some bugs require patching multiple functions, PatcherY then performs a final ranking over *clusters*. A cluster is a set of two or more functions associated with a crash that must all be patched to fix the crash. In this final clustering phase, an LLM predominantly selects which functions are related, based on the source code of the target.

4.4.4 Code Generation

Unlike unconstrained approaches, PatcherY restricts patch generation to localized edits within candidate functions found during the root cause analysis phase. This reduces the likelihood of unintended side effects and ensures that generated code remains consistent with the surrounding implementation. The LLM translates high-level repair strategies into syntactically correct and contextually appropriate code changes.

PatcherY generates a single patch per candidate cluster, but may generate multiple patches per vulnerability if each patch is verified. PatcherY treats these patches as hypotheses that require validation rather than as final outputs.

4.4.5 Patch Validation

The final stage evaluates whether generated patches correctly fix the vulnerability while preserving intended behavior. This is particularly challenging in the absence of ground truth, as was the case in the AIxCC setting.

PatcherY performs validation using a combination of dynamic execution and adversar-

ial testing. First, PatcherY applies each candidate patch and evaluates it against the original proof-of-vulnerability to ensure that the crash is no longer reproducible. This establishes that the patch addresses the observed failure.

Eliminating the crash alone is insufficient. To detect incomplete or incorrect fixes, PatcherY attempts to bypass the patch by generating new inputs that exercise similar execution paths through fuzzing [42]. If PatcherY finds a new crashing input, it rejects the patch. This adversarial validation step increases confidence that the patch addresses the underlying issue rather than masking a specific input.

PatcherY also evaluates patches against general execution behavior to detect unintended side effects through invariance checks [37]. We used invariance checks throughout the semi-finals but ultimately discarded them in the finals due to their ineffectiveness. PatcherY also used test cases when available, though they were largely unavailable in unknown targets.

While PatcherY cannot guarantee full semantic equivalence, combining multiple validation signals allows it to reject a large class of incorrect patches.

4.4.6 Discussion

This staged design reflects a key trade-off in automated repair. Unconstrained generation enables flexibility but makes correctness difficult to assess. Highly constrained approaches improve reliability but limit the space of possible fixes. PatcherY balances these by structuring the repair process: it employs LLMs where reasoning is required, while program analysis and validation constrain how that reasoning applies.

By decomposing patching into observable stages, PatcherY enables intermediate inspection and reduces variability in outcomes. This structure is particularly important in settings such as AIxCC, where systems must produce patches without direct feedback on correctness and must remain robust to adversarial evaluation.

Figure 4.2: PatcherY results across competition phases on the Nginx AIxCC CPVs. Each column corresponds to one vulnerability. A lightly filled cell indicates that a proof-of-vulnerability was available to the patcher. A darker inset with a checkmark indicates that PatcherY produced a correct patch. *Finals** evaluates the same Nginx dataset after removing the Git history (Gitless), producing a harder setting for patching. *Finals*$^{+}$ extends PatcherY with agentic-style patching (Agentic, Gitless), recovering the final two patches, yielding 14 of 14 correct patches; we discuss these additional capabilities in Section 4.6.

4.5 Evaluation

We evaluated PatcherY empirically through its deployment in the DARPA AI Cyber Challenge (AIxCC). The competition provided a realistic and adversarial setting, requiring fully autonomous systems to generate patches for vulnerabilities in real-world software without access to ground truth during execution.

Figure 4.2 summarizes PatcherY's results across competition phases. Our evaluation spans two phases of the competition: the semi-finals (Year 1) and the finals (Year 2). The finals phase further divides into two settings: one in which the organizers removed version history (*Gitless*), and one in which we added agentic capabilities to PatcherY (*Agentic, Gitless*). These settings differ in both available information and system capabilities. As a result, not all data is directly comparable across phases. We therefore present results in each setting and analyze how system changes impacted performance across a shared target, Nginx.

4.5.1 Semi-Finals Evaluation

During the semi-finals, we applied PatcherY to vulnerabilities identified in the `nginx` target. At this stage, the system operated without the full root cause analysis and ad-

vanced validation components described in Section 4.4. The system received six proofs-of-vulnerability (PoVs), each of which included a crashing input and harness source corresponding to a distinct vulnerability.

Of these six PoVs, PatcherY successfully generated correct patches for five. In the remaining case, PatcherY did not produce a patch. Importantly, PatcherY did not generate any incorrect patches in this setting. Failures stemmed from an inability to produce a valid patch rather than from producing an invalid one.

While the total number of vulnerabilities in the target was 14, we evaluated the system only on the six PoVs it received. Within this constrained setting, the high success rate suggests that PatcherY was effective at translating concrete vulnerability instances into correct patches, even without full root cause reasoning or strong validation.

4.5.2 Finals Evaluation

In the finals, we evaluated PatcherY under more challenging conditions. In the *Gitless* setting, the organizers removed all version history and commit metadata, requiring PatcherY to operate without any information about how the organizers introduced the vulnerabilities. This significantly increased the difficulty of both localization and repair, as PatcherY had to reason over the entire codebase without guidance from commit structure.

In this setting, PatcherY successfully generated correct patches for 12 of the 14 vulnerabilities in the `nginx` dataset. Compared to the semi-finals, this result reflects both the increased difficulty of the task and the expanded scope of evaluation, as PatcherY now operated on all vulnerabilities rather than on a subset defined by provided PoVs.

We further evaluated an extended version of PatcherY that incorporated agentic-style patching capabilities, enabling more flexible exploration and refinement of candidate fixes. In this *Agentic, Gitless* setting, PatcherY successfully generated correct patches for all 14 vulnerabilities.

Our final system, Artiphishell, which incorporated PatcherY as a core patching component, placed 5th in the AIxCC Finals competition. Across all submissions, the competition evaluation accepted every PatcherY-involved patch that Artiphishell submitted, yielding the highest patch accuracy among competing systems. In addition to the competition benchmarks, Artiphishell identified six previously unknown vulnerabilities in real-world software. Of these, PatcherY successfully generated verified patches for four, demonstrating its ability to generalize beyond the structured competition setting.

4.5.3 Analysis

Comparing results across phases highlights several key observations. First, the transition from semi-finals to finals shows that access to structured information, such as commit history, substantially simplifies vulnerability repair. Once the Gitless setting removes this information, the burden shifts to root cause analysis, making localization more difficult and reducing initial performance.

Second, improvements in system design, particularly the integration of stronger root cause analysis and validation, enable PatcherY to recover this gap. The improvement from 12/14 to 14/14 in the Gitless setting is primarily due to the addition of agentic capabilities, which allow PatcherY to iteratively refine candidate patches and validate them against program behavior.

Finally, these results suggest that the primary challenge in automated repair is not generating candidate patches, but ensuring their correctness. Accurate localization and effective validation have a greater impact on overall performance than unconstrained code generation. Structuring the repair process, as in PatcherY, improves reliability and allows the system to maintain high accuracy even in more difficult settings.

4.6 Extending PatcherY for Agentic Systems

While the staged design of PatcherY provides strong control and validation, it introduces limitations in flexibility when handling more complex patching scenarios. In particular, highly structured workflows struggle in cases where the repair requires broader exploration of the codebase or more iterative reasoning across multiple components.

To address this, we explored extensions to PatcherY through a more agentic approach, which we developed in collaboration with the broader Artiphishell system. This extension, called PatcherQ, leverages the same underlying tools and analyses as PatcherY, but operates within a more flexible, agent-driven framework. In this setting, PatcherQ iteratively explores code, refines hypotheses, and generates patches in a less constrained manner, similar to modern LLM-based coding systems.

Empirically, we found that combining the two approaches yielded the strongest performance. PatcherY excels in deterministic settings, where structured reasoning and validation enable efficient and reliable patch generation, but is less effective in scenarios requiring broader exploration. In contrast, PatcherQ is more flexible but incurs higher cost and variability. As a result, we adopt a hybrid strategy: we apply PatcherY first to capture straightforward cases efficiently, then fall back to PatcherQ to handle the remaining complex instances. This combination enables both efficiency and coverage.

These results suggest a natural direction for future work. Rather than treating agentic systems as a secondary fallback, developing techniques that improve the stability and verifiability of agentic workflows directly is a more effective path. Such approaches would retain the flexibility of agentic systems while achieving the reliability currently provided by structured designs like PatcherY.

4.7 Discussion and Limitations

4.7.1 Discussion

PatcherY demonstrates that highly structured, verification-driven program repair is both effective and reliable in real-world vulnerability settings. By decomposing patch generation into well-defined stages (root cause analysis, constrained code generation, and rigorous validation), PatcherY avoids many of the pitfalls of unconstrained program synthesis systems. In particular, the emphasis on validation ensures that generated patches are functionally correct with respect to the provided inputs and robust to unintended side effects.

A key takeaway from PatcherY is that determinism and structure are effective design principles in automated patching. By limiting the search space and enforcing strong correctness checks, PatcherY achieves high precision and predictability. This is especially important in security-critical contexts, where incorrect patches introduce new vulnerabilities or destabilize systems. PatcherY's modular design enables principled reasoning about failure cases, making diagnosis and improvement of individual components easier.

This structured approach introduces a fundamental trade-off. While PatcherY excels in well-defined scenarios with clear root causes and verifiable fixes, it is inherently less flexible when faced with complex, ambiguous, or multi-faceted vulnerabilities. In these cases, the strict pipeline fails to explore sufficiently diverse repair strategies, limiting overall effectiveness.

4.7.2 Limitations

Despite its strong performance, PatcherY has several important limitations:

Limited Flexibility in Complex Scenarios. PatcherY relies on deterministic pipelines and predefined transformations, which struggle with vulnerabilities requiring creative or

non-local reasoning. For example, bugs involving subtle semantic misunderstandings, cross-function interactions, or missing functionality fall outside the scope of its repair capabilities.

Dependence on Accurate Root Cause Analysis. The success of PatcherY depends heavily on the quality of its root cause analysis. If the system misidentifies the underlying issue, subsequent stages are unlikely to recover, since they will constrain the generated patches around an incorrect hypothesis. This creates a bottleneck in which early-stage errors propagate through the pipeline.

Verification Constraints. While strong validation is a strength, it is also a limitation. PatcherY only verifies properties expressible within its testing and validation framework. As a result, patches risk overfitting to available test cases or failing to generalize to unseen behaviors, particularly in the absence of comprehensive specifications.

Limited Exploration of the Search Space. Compared to more exploratory or stochastic approaches, PatcherY intentionally restricts the space of candidate patches. While this improves efficiency and correctness, the restriction excludes valid but unconventional repairs, particularly in edge cases.

4.7.3 Implications

These limitations highlight a broader insight: no single paradigm is sufficient for automated program repair. Deterministic, verification-heavy systems like PatcherY provide reliability and precision, while more flexible, agentic systems offer broader exploration and adaptability. This observation motivates hybrid approaches that combine the strengths of both paradigms.

In particular, as discussed in Section 4.6, integrating PatcherY with more flexible, agentic systems enables a staged approach: leveraging PatcherY for fast, reliable repairs in well-structured cases, and deferring to more expressive systems when additional reasoning is required. This combination represents a promising direction toward more general and effective automated patching systems.

4.8 Related Work

Automated program repair (APR) has been a long-standing goal in software engineering and security research. Prior work spans a spectrum of approaches, from search-based and template-driven systems to learning-based and, more recently, agentic methods. PatcherY builds on this body of work by emphasizing structured repair and strong validation in security-critical contexts.

4.8.1 Search-Based and Template-Based Repair

Early APR systems focused on search-based techniques that explore a space of candidate patches guided by test suites. Systems such as GenProg [65], SPR [69], and Prophet [68] demonstrated that it is possible to automatically repair real-world bugs by mutating existing code and selecting patches that pass all tests. Template-based approaches further constrained this search space by leveraging predefined repair patterns, improving efficiency and interpretability.

While these methods showed promising results, they suffer from well-documented limitations. In particular, they often generate patches that overfit to test suites without addressing the underlying root cause of a bug [88]. Additionally, their reliance on syntactic transformations limits their ability to handle complex semantic errors. PatcherY differs from these approaches by explicitly modeling root cause analysis and incorporating structured validation, reducing the likelihood of overfitting and improving patch correctness.

4.8.2 Semantic and Constraint-Based Repair

To address the limitations of purely syntactic methods, subsequent work introduced semantic and constraint-based repair techniques. Systems such as SemFix [81] and Angelix [75] use symbolic execution and constraint solving to synthesize patches that satisfy formal specifications derived from program behavior. These approaches provide stronger correctness guarantees compared to search-based methods, as they reason about program semantics rather than surface-level syntax.

Scalability challenges and the difficulty of generating accurate specifications often limit semantic repair systems. In practice, they struggle with large codebases or complex program states. PatcherY adopts a similar philosophy of correctness-driven repair but avoids heavy reliance on constraint solving, instead combining lightweight analysis with practical validation strategies that scale to real-world systems.

4.8.3 Learning-Based Program Repair

More recent work has explored the use of machine learning for program repair. Researchers have applied neural models to tasks such as bug localization, patch generation, and ranking, often leveraging large corpora of code to learn repair patterns. Large language models (LLMs), in particular, have demonstrated the ability to generate plausible patches directly from natural language descriptions or code context [60].

Despite their flexibility, learning-based approaches introduce new challenges. Generated patches are sometimes syntactically correct but semantically incorrect, and models often lack strong guarantees of correctness. In addition, their behavior is difficult to interpret and control, which is problematic in security-sensitive settings. PatcherY contrasts with these approaches by prioritizing determinism and verification, ensuring that generated patches meet explicit correctness criteria before deployment.

4.9 Conclusion

In this chapter, we presented PatcherY, a structured and verification-driven system for automated vulnerability repair. By combining root cause analysis, constrained code generation, and rigorous validation, PatcherY demonstrates that reliable automated patching is achievable in real-world security settings.

More broadly, PatcherY highlights that automated patching is fundamentally a problem of understanding. These systems must reason about software they did not write, often without specifications or developer intent. This places automated repair squarely within the domain of reverse engineering: reconstructing semantics and intent from incomplete artifacts. In this way, PatcherY operationalizes automated reverse engineering as a prerequisite for autonomous action.

This perspective reframes autonomy in software systems. The ability to act is inseparable from the ability to understand. Without understanding, automated systems risk producing brittle or incorrect results. PatcherY shows that grounding repair in structured analysis and validation provides this understanding, enabling more reliable outcomes.

Ultimately, advancing autonomous vulnerability repair requires deeper integration between reverse engineering and automation. PatcherY represents a step in this direction, demonstrating that understanding-driven approaches are central to building safe and effective autonomous systems.

Chapter 5

CONCLUSION

Software reverse engineering has long been an essential, yet loosely defined, discipline. As software systems grow in complexity and are increasingly developed, transformed, and even generated without full human oversight, the need to understand and act on code without complete context has become fundamental. In this dissertation, I argued that reverse engineering can be transformed into a science by systematically studying its techniques, processes, and applications. I approached this goal through three complementary contributions.

First, I studied reverse engineering techniques by addressing the lack of principled metrics for evaluating their quality. Through SAILR, I introduced closeness-to-source as a rigorous goal for decompilation, along with Graph Edit Distance as its quantitative metric, and demonstrated that principled metrics reveal how techniques must be redesigned to achieve their goals. By identifying compiler transformations as a primary source of decompilation error and explicitly inverting them, SAILR showed that advancing individual techniques requires both defining what success means and building systems that achieve it.

Second, I studied the reverse engineering process, examining how analysts compose multiple techniques into effective workflows. Through an empirical study of human and LLM-assisted workflows, I showed that the process of reverse engineering is dynamic and strategy-dependent, shaped by how analysts iteratively select and combine techniques. These findings highlight the importance of studying reverse engineering not only through individual techniques but through the processes that connect them.

Third, I studied the applications of reverse engineering, which ultimately define the utility of our techniques and metrics. With PatcherY, I showed that application demands,

such as automated vulnerability repair, provide a concrete test of whether reverse engineering methods serve real-world needs. My results indicate that studying applications reveals which aspects of understanding are essential for successful intervention and which are not, informing the design of future techniques.

Together, these contributions frame reverse engineering through a unified lens: a discipline whose advancement requires studying techniques, processes, and applications in concert. By connecting these three aspects, this dissertation establishes a foundation for reasoning about reverse engineering in a more principled and coherent manner.

Reverse engineering is no longer confined to specialized settings such as binary analysis. It is becoming a fundamental aspect of interacting with modern software systems. By systematically studying its techniques, processes, and applications, I take a step toward transforming reverse engineering from a collection of ad hoc methods into a coherent and systematic discipline.

I hope that this perspective encourages future work that continues to refine, formalize, and expand the science of software reverse engineering.

REFERENCES

[1] "Debian popularity contest statistics: Installations", https://popcon.debian.org/by_inst (2023).

[2] "GIMPLE (GNU compiler collection (GCC) internals)", https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html (2023).

[3] "Hex-Rays – state of the art binary analysis solutions", https://www.hex-rays.com/ (2023).

[4] "Martin Liska: Switch lowering improvements – slideslive", https://slideslive.com/38902416/switch-lowering-improvements (2023).

[5] Allen, F. E., "Control flow analysis", in "Proceedings of the ACM Symposium on Compiler Optimization", (1970).

[6] Aonzo, S., Y. Han, A. Mantovani and D. Balzarotti, "Humans vs. machines in malware classification", in "32nd USENIX Security Symposium (USENIX Security 23)", pp. 1145–1162 (USENIX Association, Anaheim, CA, 2023), URL https://www.usenix.org/conference/usenixsecurity23/presentation/aonzo.

[7] Arcuri, A. and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering", Software Testing, Verification and Reliability **24**, 3, 219–250 (2014).

[8] Artuso, F., G. A. Di Luna, L. Massarelli and L. Querzoni, "Function naming in stripped binaries using neural networks", arXiv preprint arXiv:1912.07946 (2019).

[9] Atredis Partners, "aiDAPal", https://github.com/atredispartners/aidapal (Accessed May 11, 2026).

[10] Basque, Z. L., A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili and R. Wang, "Ahoy sailr! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation", in "33rd USENIX Security Symposium (USENIX Security 24)", pp. 361–378 (2024).

[11] Benjamini, Y. and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing", Journal of the Royal statistical society: series B (Methodological) **57**, 1, 289–300 (1995).

[12] Binary Ninja, "Binary Ninja Sidekick", https://sidekick.binary.ninja/ (Accessed May 11, 2026).

[13] Börstler, J. and B. Paech, "The role of method chains and comments in software readability and comprehension—an experiment", IEEE Transactions on Software Engineering **42**, 9, 886–898 (2016).

[14] Bourquin, M., A. King and E. Robbins, "Binslayer: accurate comparison of binary executables", in "Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop", pp. 1–10 (2013).

[15] Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners", Advances in neural information processing systems **33**, 1877–1901 (2020).

[16] Brumley, D., I. Jager, T. Avgerinos and E. J. Schwartz, "Bap: A binary analysis platform", in "International Conference on Computer Aided Verification", pp. 463–469 (Springer, 2011).

[17] Brumley, D., J. Lee, E. J. Schwartz and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring", in "22nd USENIX Security Symposium (USENIX Security 13)", pp. 353–368 (2013).

[18] Bunke, H., "On a relation between graph edit distance and maximum common subgraph", Pattern recognition letters **18**, 8, 689–694 (1997).

[19] Burk, K., F. Pagani, C. Kruegel and G. Vigna, "Decomperson: How humans decompile and what we can learn from it", in "USENIX Security Symposium", (2022).

[20] Chen, Q., J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig and B. Vasilescu, "Augmenting Decompiler Output with Learned Variable Names and Types", in "Proceedings of the USENIX Security Symposium", (2022).

[21] Chen, Q., J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning", arXiv preprint arXiv:2112.02650 (2021).

[22] Cifuentes, C., *Reverse compilation techniques* (Citeseer, 1994).

[23] Cifuentes, C. and K. J. Gough, "Decompilation of binary programs", Software: Practice and Experience **25**, 7, 811–829 (1995).

[24] Cliff, N., "Dominance statistics: Ordinal analyses to answer ordinal questions.", Psychological bulletin **114**, 3, 494 (1993).

[25] Cocke, J., "Global common subexpression elimination", in "Proceedings of the ACM Symposium on Compiler Optimization", (1970).

[26] Cohen, J., *Statistical power analysis for the behavioral sciences* (routledge, 2013).

[27] Cohen, J., "A power primer", Methodological Issues and Strategies in Clinical Research pp. 279–284 (2016).

[28] DARPA, "AI Cyber Challenge (AIxCC)", `https://www.darpa.mil/research/programs/ai-cyber` (2024).

[29] Deng, G., Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger and S. Rass, "PentestGPT: Evaluating and harnessing large language models for automated penetration testing", in "33rd USENIX Security Symposium (USENIX Security 24)", pp. 847–864 (USENIX Association, Philadelphia, PA, 2024), URL https://www.usenix.org/conference/usenixsecurity24/presentation/deng.

[30] Di Federico, A., P. Fezzardi and G. Agosta, "rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery", in "2018 International Carnahan Conference on Security Technology (ICCST)", pp. 1–5 (IEEE, 2018).

[31] Dijkstra, E. W., "Letters to the editor: go to statement considered harmful", Communications of the ACM **11**, 3, 147–148 (1968).

[32] Ďurfina, L., J. Křoustek and P. Zemek, "Psybot malware: A step-by-step decompilation case study", in "2013 20th Working Conference on Reverse Engineering (WCRE)", pp. 449–456 (IEEE, 2013).

[33] Ďurfina, L., J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík and A. Meduna, "Design of a retargetable decompiler for a static platform-independent malware analysis", in "International Conference on Information Security and Assurance", pp. 72–86 (Springer, 2011).

[34] Eghbal, N., "Roads and bridges: The unseen labor behind our digital infrastructure", Tech. rep., Ford Foundation (2016).

[35] Ellard-Gray, A., N. K. Jeffrey, M. Choubak and S. E. Crann, "Finding the hidden participant: Solutions for recruiting hidden, hard-to-reach, and vulnerable populations", International journal of qualitative methods **14**, 5, 1609406915621420 (2015).

[36] Engel, F., R. Leupers, G. Ascheid, M. Ferger and M. Beemster, "Enhanced structural analysis for c code reconstruction from ir code", in "Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems", pp. 21–27 (2011).

[37] Ernst, M. D., J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, "The daikon system for dynamic detection of likely invariants", Science of computer programming **69**, 1-3, 35–45 (2007).

[38] Erosa, A. M. and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements", in "Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)", pp. 229–240 (IEEE, 1994).

[39] Evyatar E., "GptHidra", https://github.com/evyatar9/GptHidra (Accessed May 11, 2026).

[40] Fagerland, M. W. and L. Sandvik, "Performance of five two-sample location tests for skewed distributions with unequal variances", Contemporary clinical trials **30**, 5, 490–496 (2009).

[41] Fay, M. P. and M. A. Proschan, "Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules", Statistics surveys **4**, 1 (2010).

[42] Fioraldi, A., D. Maier, H. Eißfeldt and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research", in "14th USENIX workshop on offensive technologies (WOOT 20)", (2020).

[43] Ford, I., A. Soneji, F. B. Kokulu, J. Vadayath, Z. L. Basque, G. Vipat, A. Doupé, R. Wang, G.-J. Ahn, T. Bao *et al.*, ""watching over the shoulder of a professional": Why hackers make mistakes and how they fix them", in "2024 IEEE Symposium on Security and Privacy (SP)", pp. 350–368 (IEEE, 2024).

[44] Fu, C., H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar and J. Zhao, "A neural-based program decompiler", arXiv:1906.12029 [cs] URL `http://arxiv.org/abs/1906.12029`, arXiv: 1906.12029 (2019).

[45] Funder, D. C. and D. J. Ozer, "Evaluating effect size in psychological research: Sense and nonsense", Advances in methods and practices in psychological science **2**, 2, 156–168 (2019).

[46] Gastwirth, J. L., Y. R. Gel and W. Miao, "The impact of levene's test of equality of variances on statistical theory and practice", Statistical Science **24**, 3, 343–360 (2009).

[47] Gibbons, J. D. and S. Chakraborti, *Nonparametric statistical inference: revised and expanded* (CRC press, 2014).

[48] Google Project Zero, "From Naptime to Big Sleep: Using Large Language Models To Catch Vulnerabilities In Real-World Code", `https://googleprojectzero.blogspot.com/2024/10/from-naptime-to-big-sleep.html` (Accessed May 11, 2026).

[49] Grissom, R. J. and J. J. Kim, *Effect sizes for research: Univariate and multivariate applications* (Routledge, 2012).

[50] Gussoni, A., A. Di Federico, P. Fezzardi and G. Agosta, "A comb for decompiled c code", in "Proceedings of the 15th ACM Asia Conference on Computer and Communications Security", pp. 637–651 (2020).

[51] Halstead, M. H., *Machine-independent computer programming* (Spartan Books, 1962).

[52] Hardekopf, B. and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code", in "Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation", pp. 290–299 (2007).

[53] Hart, A., "Mann-whitney test is not just a test of medians: differences in spread can be important", Bmj **323**, 7309, 391–393 (2001).

[54] He, J., 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", pp. 1667–1680 (2018).

[55] Hedges, L. V., "Distribution theory for glass's estimator of effect size and related estimators", journal of Educational Statistics **6**, 2, 107–128 (1981).

[56] Hu, P., R. Liang and K. Chen, "Degpt: Optimizing decompiler output with llm", in "Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID", vol. 267622140 (2024).

[57] Ivan Kwiatkowski, "Gepetto", `https://github.com/JusticeRage/Gepetto` (Accessed May 11, 2026).

[58] Jaffe, A., J. Lacomis, E. J. Schwartz, C. L. Goues and B. Vasilescu, "Meaningful variable names for decompiled code: A machine translation approach", in "Proceedings of the 26th Conference on Program Comprehension", pp. 20–30 (2018).

[59] Jiang, L., X. Jin and Z. Lin, "Beyond classification: Inferring function names in stripped binaries via domain adapted llms", in "Proceedings 2025 Network and Distributed System Security Symposium (2025). https://api. semanticscholar. org/CorpusID", (2025).

[60] Jimenez, C. E., J. Yang, A. Wettig, S. Yao, K. Pei, O. Press and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?", in "The Twelfth International Conference on Learning Representations", (2024), URL `https://openreview.net/forum?id=VTF8yNQM66`.

[61] Kampenes, V. B., T. Dybå, J. E. Hannay and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments", Information and Software Technology **49**, 11-12, 1073–1086 (2007).

[62] Kernel, L., "Linux Kernel", `https://github.com/torvalds/linux/tree/master/kernel` (2023).

[63] Knuth, D. E., J. H. Morris, Jr and V. R. Pratt, "Fast pattern matching in strings", SIAM journal on computing **6**, 2, 323–350 (1977).

[64] Lacomis, J., P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig and B. Vasilescu, "DIRE: A neural approach to decompiled identifier naming", in "2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)", pp. 628–639 (IEEE, 2019).

[65] Le Goues, C., T. Nguyen, S. Forrest and W. Weimer, "Genprog: A generic method for automatic software repair", Ieee transactions on software engineering **38**, 1, 54–72 (2011).

[66] Lee, J., T. Avgerinos and D. Brumley, "Tie: Principled reverse engineering of types in binary programs", in "Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)", p. 18 (2011).

[67] Li, Z., S. Dutta and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities", arXiv preprint arXiv:2405.17238 (2024).

[68] Long, F. and M. Rinard, "Automatic patch generation by learning correct code", in "Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages", pp. 298–312 (2016).

[69] Long, F. and M. C. Rinard, "Staged program repair with condition synthesis", in "Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)", pp. 166–178 (2015).

[70] Mantovani, A., S. Aonzo, Y. Fratantonio and D. Balzarotti, "RE-Mind: a first look inside the mind of a reverse engineer", in "31st USENIX Security Symposium (USENIX Security 22)", pp. 2727–2745 (USENIX Association, Boston, MA, 2022), URL https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani.

[71] Mantovani, A., S. Aonzo, Y. Fratantonio and D. Balzarotti, "RE-Mind: a first look inside the mind of a reverse engineer", in "31st USENIX Security Symposium (USENIX Security 22)", pp. 2727–2745 (USENIX Association, Boston, MA, 2022), URL https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani.

[72] Mantovani, A., L. Compagna, Y. Shoshitaishvili and D. Balzarotti, "The convergence of source code and binary vulnerability discovery–a case study", in "Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security", pp. 602–615 (2022).

[73] Marfo, P. and G. Okyere, "The accuracy of effect-size estimates under normals and contaminated normals in meta-analysis", Heliyon **5**, 6 (2019).

[74] Mattias Karlsson, "GhidraChatGPT", https://github.com/likvidera/GhidraChatGPT (Accessed May 11, 2026).

[75] Mechtaev, S., J. Yi and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis", in "Proceedings of the 38th International Conference on Software Engineering (ICSE)", pp. 691–701 (2016).

[76] Mei, X., P. S. Singaria, J. Del Castillo, H. Xi, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, H. Pearce, B. Dolan-Gavitt *et al.*, "Arvo: Atlas of reproducible vulnerabilities for open source software", arXiv preprint arXiv:2408.02153 (2024).

[77] Mirzaei, O., R. Vasilenko, E. Kirda, L. Lu and A. Kharraz, "Scrutinizer: Detecting code reuse in malware via decompilation and machine learning", in "International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment", pp. 130–150 (Springer, 2021).

[78] Nam, D., A. Macvean, V. Hellendoorn, B. Vasilescu and B. Myers, "Using an llm to help with code understanding", in "Proceedings of the IEEE/ACM 46th International Conference on Software Engineering", ICSE '24 (Association for Computing

Machinery, New York, NY, USA, 2024), URL https://doi.org/10.1145/3597503.3639187.

[79] National Security Agency (NSA), "VirusTotal", https://ghidra-sre.org/ (Accessed May 11, 2026).

[80] Nelson, C. and Y. Shoshitaishvili, "Dojo: Applied cybersecurity education in the browser", in "Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1", pp. 930–936 (2024).

[81] Nguyen, H. D. T., D. Qi, A. Roychoudhury and S. Chandra, "SemFix: Program repair via semantic analysis", in "Proceedings of the 35th International Conference on Software Engineering (ICSE)", pp. 772–781 (2013).

[82] Nguyen, S., H. M. Babe, Y. Zi, A. Guha, C. J. Anderson and M. Q. Feldman, "How beginning programmers and code llms (mis)read each other", in "Proceedings of the CHI Conference on Human Factors in Computing Systems", vol. 1 of *CHI '24*, p. 1–26 (ACM, 2024), URL http://dx.doi.org/10.1145/3613904.3642706.

[83] NIST, "CVE-2024-3094: XZ Utils backdoor", https://nvd.nist.gov/vuln/detail/CVE-2024-3094 (2024).

[84] Noonan, M., A. Loginov and D. Cok, "Polymorphic type inference for machine code", in "Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation", p. 27–41 (2016), URL http://arxiv.org/abs/1603.05495.

[85] Nosco, T., J. Ziegler, Z. Clark, D. Marrero, T. Finkler, A. Barbarello and W. M. Petullo, "The industrial age of hacking", in "29th USENIX Security Symposium (USENIX Security 20)", pp. 1129–1146 (2020).

[86] Park, Y., H. Lee, J. Jung, H. Koo and H. K. Kim, "Benzene: A practical root cause analysis system with an under-constrained state mutation", in "2024 IEEE Symposium on Security and Privacy (SP)", pp. 1865–1883 (IEEE, 2024).

[87] Pearce, H., B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri and B. Dolan-Gavitt, "Pop quiz! can a large language model help with reverse engineering?", URL https://arxiv.org/abs/2202.01142 (2022).

[88] Qi, Z., F. Long, S. Achour and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems", in "Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)", pp. 24–36 (2015).

[89] Radford, A., J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners", OpenAI blog **1**, 8, 9 (2019).

[90] Reiter, P., H. J. Tay, W. Weimer, A. Doupé, R. Wang and S. Forrest, "Automatically mitigating vulnerabilities in x86 binary programs via partially recompilable decompilation", arXiv preprint arXiv:2202.12336 (2022).

[91] Rukmono, S. A., L. Ochoa and M. R. Chaudron, "Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis", in "2023 IEEE International Conference on Data and Software Engineering (ICoDSE)", pp. 7–12 (IEEE, 2023).

[92] Shang, X., S. Cheng, G. Chen, Y. Zhang, L. Hu, X. Yu, G. Li, W. Zhang and N. Yu, "How far have we gone in binary code understanding using large language models", in "2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)", pp. 1–12 (2024).

[93] Sharir, M., "Structural analysis: A new approach to flow analysis in optimizing compilers", Computer Languages **5**, 3-4, 141–153 (1980).

[94] Sheng, Z., Z. Chen, S. Gu, H. Huang, G. Gu and J. Huang, "Llms in software security: A survey of vulnerability detection techniques and insights", URL `https://arxiv.org/abs/2502.07049` (2025).

[95] Shoshitaishvili, Y., 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)", pp. 138–157 (IEEE, 2016).

[96] Slowinska, A., T. Stancescu and H. Bos, "Howard: a dynamic excavator for reverse engineering data structures", in "Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)", (2011).

[97] SpyEye, "SpyEye", `https://github.com/ytisf/theZoo/tree/master/malware/Binaries/SpyEye` (2023).

[98] Thoppilan, R., D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du *et al.*, "Lamda: Language models for dialog applications", arXiv preprint arXiv:2201.08239 (2022).

[99] Tim Blazytko, "ReverserAI", `https://github.com/mrphrazer/reverser_ai` (Accessed May 11, 2026).

[100] Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention is all you need", Advances in neural information processing systems **30** (2017).

[101] Votipka, D., S. Rabin, K. Micinski, J. S. Foster and M. L. Mazurek, "An observational investigation of reverse Engineers' processes", in "29th USENIX Security Symposium (USENIX Security 20)", pp. 1875–1892 (USENIX Association, 2020), URL `https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational`.

[102] Williams, M. H. and G. Chen, "Restructuring pascal programs containing goto statements", The Computer Journal **28**, 2, 134–137 (1985).

[103] Xie, D., Z. Zhang, N. Jiang, X. Xu, L. Tan and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries", in "Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security", pp. 4554–4568 (2024).

[104] Xu, X., 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", CCS '17, pp. 363—-376 (Association for Computing Machinery, New York, NY, USA, 2017), URL https://doi.org/10.1145/3133956.3134018.

[105] Xu, X., Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan *et al.*, "Unleashing the power of generative model in recovering variable names from stripped binary", in "Proceedings of the Network and Distributed System Security Symposium (NDSS)", (2025).

[106] Yakdan, K., S. Dechand, E. Gerhards-Padilla and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study", in "2016 IEEE Symposium on Security and Privacy (SP)", pp. 158–177 (IEEE, 2016).

[107] Yakdan, K., S. Eschweiler, E. Gerhards-Padilla and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations", in "Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)", (2015).

[108] Yamaguchi, F., C. Wressnegger, H. Gascon and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery", in "Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security", pp. 499–510 (2013).

[109] Yang, J., C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering", in "Advances in Neural Information Processing Systems (NeurIPS)", (2024).

[110] Zamfirescu-Pereira, J., R. Y. Wong, B. Hartmann and Q. Yang, "Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts", in "Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems", CHI '23 (Association for Computing Machinery, New York, NY, USA, 2023), URL https://doi.org/10.1145/3544548.3581388.

[111] Zhang, R., N. J. McNeese, G. Freeman and G. Musick, ""an ideal human" expectations of ai teammates in human-ai teaming", Proceedings of the ACM on Human-Computer Interaction **4**, CSCW3, 1–25 (2021).

[112] Zhang, T. T., C. Taylor, B. Coppens, W. Mebane, C. Collberg and B. De Sutter, "reanalyst: Scalable annotation of reverse engineering activities", Journal of Systems and Software p. 112492 (2025).

[113] Zhang, Y., H. Ruan, Z. Fan and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement", in "Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)", (2024).

[114] Zhang, Z., Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary", in "2021 IEEE Symposium on Security and Privacy (SP)", p. 813–832 (IEEE, 2021), URL https://ieeexplore.ieee.org/document/9519451/.

APPENDIX A

PERMISSION FROM CO-AUTHORS

Permission has been obtained from all co-authors for use of the following published works in this dissertation:

- Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. "Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation". In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 361–378.

- Zion Leonahenahe Basque, Samuele Doria, Ananta Soneji, Wil Gibbs, Adam Doupé, Yan Shoshitaishvili, Eleonora Losiouk, Ruoyu Wang, and Simone Aonzo. "Decompiling the Synergy: An Empirical Study of Human–LLM Teaming in Software Reverse Engineering". In: *Proceedings of the 2026 Network and Distributed System Security Symposium (NDSS 2026)*. 2026.