Methods and Tools for Practical Software Testing and Maintenance

Anthony Saieva Narin

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy under the Executive Committee of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2024
Abstract

Methods and Tools for Practical Software Testing and Maintenance

Anthony Saieva

As software continues to envelop traditional industries the need for increased attention to cybersecurity is higher than ever. Software security helps protect businesses and governments from financial losses due to cyberattacks and data breaches, as well as reputational damage. In theory, securing software is relatively straightforward—it involves following certain best practices and guidelines to ensure that the software is secure. In practice, however, software security is often much more complicated. It requires a deep understanding of the underlying system and code (including potentially legacy code), as well as a comprehensive understanding of the threats and vulnerabilities that could be present. Additionally, software security also involves the implementation of strategies to protect against those threats and vulnerabilities, which may involve a combination of technologies, processes, and procedures. In fact many real cyber attacks are caused not from zero day vulnerabilities but from known issues that haven’t been addressed so real software security also requires ongoing monitoring and maintenance to ensure critical systems remain secure.

This thesis presents a series of novel techniques that together form an enhanced software maintenance methodology from initial bug reporting all the way through patch deployment. We begin by introducing Ad Hoc Test Generation, a novel testing technique that handles when a security vulnerability or other critical bug is not detected by the developers’ test suite, and is discovered post-deployment, developers must quickly devise a new test that reproduces the buggy behavior. Then the developers need to test whether their candidate patch indeed fixes the bug, without breaking other
functionality, while racing to deploy before attackers pounce on exposed user installations. This work builds on record-replay and binary rewriting to automatically generate and run targeted tests for candidate patches significantly faster and more efficiently than traditional test suite generation techniques like symbolic execution. Our prototype of this concept is called ATTUNE.

To construct patches in some instances developers maintaining software may be forced to deal directly with the binary since source code is no longer available. In these instances this work presents a transformer based model called DIRECT that provides semantics related names for variables and function names that have been lost giving developers the opportunity to work with a facsimile of the source code that would otherwise be unavailable. In the event developers need even more support deciphering the decompiled code we provide another tool called REINFOREST that allows developers to search for similar code which they can use to further understand the code in question and use as a reference when developing a patch.

After patches have been written, deployment remains a challenge. In some instances deploying a patch for the buggy behavior may require supporting legacy systems where software cannot be upgraded without causing compatibility issues. To support these updates this work introduces the concept of binary patch decomposition which breaks a software release down into its component parts and allows software administrators to apply only the critical portions without breaking functionality.

We present a novel software patching methodology that we can recreate bugs, develop patches, and deploy updates in the presence of the typical challenges that come when patching production software including deficient test suites, lack of source code, lack of documentation, compatibility issues, and the difficulties associated with patching binaries directly.
# Table of Contents

Acknowledgments ................................................................. vi

Dedication ............................................................................... vii

Preface ..................................................................................... 1

Chapter 1: Introduction ............................................................ 2

1.1 Motivation ........................................................................... 2

1.2 Background ........................................................................ 3

1.3 Contribution – A new update paradigm .............................. 4

1.4 Thesis Statement ............................................................... 8

Chapter 2: Related Work ............................................................ 9

2.1 Traditional Procedures ...................................................... 9

2.1.1 Test generation .......................................................... 9

2.1.2 Dynamic debugging techniques - MVE and Fuzzing ....... 10

2.1.3 Live Instrumentation and Debugging support ............... 11

2.1.4 Looking Forward ....................................................... 11

2.2 Record/Replay Validation .................................................. 12

2.2.1 Recording .................................................................. 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>6.2</td>
<td>Background</td>
</tr>
<tr>
<td>6.2.1</td>
<td>ELF Format and Existing Binary Diffs</td>
</tr>
<tr>
<td>6.2.2</td>
<td>x86 Calling Conventions</td>
</tr>
<tr>
<td>6.3</td>
<td>Design</td>
</tr>
<tr>
<td>6.3.1</td>
<td>BPD Design</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Binary Quilting Procedure</td>
</tr>
<tr>
<td>6.4</td>
<td>Evaluation</td>
</tr>
<tr>
<td>6.5</td>
<td>Limitations</td>
</tr>
<tr>
<td>6.6</td>
<td>Conclusion and Future Work</td>
</tr>
<tr>
<td>7.1</td>
<td>Generating Unit Tests vs. Generating System Tests</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Feasibility Studies:</td>
</tr>
<tr>
<td>7.2</td>
<td>Adding Assertions for Side-Effect Detection</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Feasibility Studies:</td>
</tr>
<tr>
<td>Epilogue</td>
<td></td>
</tr>
<tr>
<td>References</td>
<td></td>
</tr>
</tbody>
</table>
Acknowledgements

First I would like to acknowledge my advisor Gail Kaiser for giving me, a stranger, a chance when nobody else would and always judging me on my merit. To the rest of my committee and other members of the Columbia Faculty, I will always appreciate the lessons you taught me both about research and independent life. I would also like to thank NSF and DARPA for their support of this research. This research was funded in part by NSF CNS-1563555, CNS-1842456 and CCF-1815494, and in part by DARPA / NIWC Pacific N66001-21-C-4018.

To my many friends and collaborators some of whom include, Preetam Dutta, David Williams King, Riley Spahn, Saikat Chakraborty, Luca Piccolboni, Robin (Yangruibo) Ding, Mike Su, Vikram Nitin, Dennis Roellke, Carmine Elvezio, and Lydia Chilton. Thank you for all your time and attention.

A special moment for Davide Giri, gone too soon, and an inspiration to me always. I am very proud to call you a friend and that you would do the same.
Dedication

To the reader.
Preface

It’s important to recognize that software security must consider both the identification of bugs as well as the ability to patch those bugs in the field. Securing software depends on both knowledge and the ability to act on it. I started on this journey after working at a large bank and seeing the difficulties we had in securing the systems there. In the process of this research I found that these issues extend far beyond the banking system and into all areas of our technology infrastructure from government to defense.

For many years we have investigated how to produce software without bugs and identify vulnerabilities before deployment. However I think we must acknowledge that as long as technology has features, it will have bugs, and attackers will exploit them. Preventing and detecting bugs is an important part of the battle, however in practice most security issues stem not from zero day vulnerabilities but instead from known problems that have not been addressed yet. In order to truly develop a robust technological infrastructure it must be one that is built to evolve over time and we must provide the tools and practices required to handle patching and deployment in practical settings for long periods of time over many software iterations and technological life-cycles.

It is my sincere hope that this work is a step towards that end.
Chapter 1: Introduction

1.1 Motivation

In the modern era of rapid technological advancement software maintenance practices in ensuring the longevity, reliability, and adaptability of software systems take on increased importance. As the lifeblood of various industries, software systems are continuously evolving to cater to the dynamic needs of end-users and to accommodate the ever-changing technological landscape. Hence, the implementation of robust software maintenance practices is paramount to the sustainability of these systems, as they not only improve the overall quality and performance but also prevent attackers from exploiting previously vulnerable systems.

The traditional software update process is outlined in Figure 1.1a. Users interact with code deployed in a production environment that is managed by onsite IT staff also called in this work an operator or administrator. Sometimes the operator is part of the same organization as the software developer but more often than not in practice we find that the software is overseen by a separate operator [87]. As such when users experience a bug in production the operator compiles a bug report and sends this to the developer. The bug report may include stack traces, videos, commentary, and screenshots attempting to provide all relevant information for the developer to remedy the situation. Unfortunately due to the complex nature of computer systems in practice, these reports do not contain all of the relevant information [7]. This only gets more complicated as production environments also increase in complexity with sophisticated cloud deployments. The developer then armed only with a bug report attempts to produce a patched version of the software. Hopefully, the developer also writes a test to accompany the patched code ensuring that future versions maintain the patched behavior, but more often than not developers fail to maintain adequate test suites [189]. Before the patched software is released most modern deployment pipelines have some
level of continuous integration (CI) and continuous deployment (CD) in which the code is built, undergoes regression testing, and gets uploaded to a remote distribution repository that contains the patched binaries that need to be deployed. Finally, the operator deploys the patched software to the production environment.

While this process is the foundation for modern software deployment, many problems arise.

### 1.2 Background

First, it is exceedingly difficult to recreate the buggy behavior and maintain adequate test suites as mentioned in the last section. Even after a patch is developed, without being able to completely recreate the production environment that caused the bug in the first place, the developer has no confidence they have successfully fixed the problem [63]. Secondly, as software ages patching becomes increasingly more difficult as languages fall out of style [139], developers change positions, documentation gets lost, and foundational technology gets replaced [121], yet the legacy software remains [11]. In many such cases, source code may even be lost and only the running binaries remain. Situations like these may sound abstract, but in practice software deployments with these exact problems power critical infrastructure around the globe including power grids, transportation systems, and air traffic control [45]. Finally, even if a patch can be produced and a patched binary can be built, distribution remains an issue as operators are hesitant to deploy new software that may break critical functionality due to issues in backward compatibility, especially with required libraries [93]. This is exasperated by the fact that software updates are usually released on a scheduled cycle including lots of changes in a single update some of which may break functionality the operator deems critical even if the update also includes patches the operator must deploy.

Software bugs in such critical allow malicious actors to cause serious damage with severe technical and real life consequences. It turns out that most cyber security incidents actually deal with known vulnerabilities and not zero-day issues [79]. Further Mink et al. [126] show that patching bugs as quickly as possible is of critical importance to mitigate damages. Solving most software security incidents in practice involves solving update and deployment procedures.
1.3 Contribution – A new update paradigm

With modern technology, we should support developers and operators trying to manage these difficult situations and this thesis suggests leveraging new technology to change the update process in a few critical ways outlined in Figure 1.1b. First modern record replay technology should alleviate the need for antiquated inefficient bug reports. Instead production level code can be recorded with lightweight recording systems like [103] and then that log can be augmented with a more heavyweight solution like [136] which will provide the developer with all the information they need to recreate the bug. This log becomes an exact test case that we call an Ad Hoc Test that can be added to existing test suites to ensure against future regressions. Still, once the bug is recreated, in the difficult circumstances outlined earlier developers need additional support. If source code has been lost modern tools should be able to improve on traditional decompilation providing semantic meaning to binaries like variable and function names. In circumstances when documentation is lost or the legacy code is written in an antiquated language, with modern techniques developers can look for similar code as a reference in languages they are more familiar with or in code bases that contain documentation. When a patch is developed using modern binary rewriting in conjunction with replay technology developers can actually verify that their patch works under the exact circumstances that triggered the bug in the production environment. Further, with the test cases collected CI/CD remains up to date without requiring developers to write additional tests or depending on metrics like code coverage to determine the adequacy of the test suite. Instead, they can rely on the actual circumstances that took place in practice with replay testing. Finally, once the patched binary is distributed with modern advances in binary rewriting that allow for complete binary recompilation [185] operators are no longer tied to a singular update that services every deployment. They can instead pick and choose customized updates that meet their needs.

To give developers the opportunity to recreate buggy executions and validate patches, this thesis begins by presenting a tool called ATTUNE that builds on top of existing record replay technology allowing developers to verify the patches they’ve created actually fix the bugs they are intended to.
Figure 1.1: Enhanced practical software update procedure
**ATTUNE** leverages binary rewriting to take a recording of the buggy execution environment and sew the patch into the buggy binary. Once the patched binary is sewn into the address space then the developer can continue to re-run the buggy execution over the patched code to guarantee that the issue is resolved. If the issue is not resolved then the developer can examine further problems with the help of a debugger and continue to produce potential patches.

Then to support developers that do not have access to source code this thesis introduces a tool called **DIRECT** that leverages modern transformer-based architectures adapted from the natural language processing (NLP) community to take decompiled binaries and assign meaningful variable names. **DIRECT** supports developers by adding semantic meaning to otherwise unintelligible binary code. **ATTUNE** and **DIRECT** work in conjunction as developers can use **DIRECT** to develop patches that are validated in **ATTUNE**. To further support developers working without source code or documentation this thesis describes a tool called **REINFOREST** that builds on top of state-of-the-art large language models and allows developers to search for similar reference code including code in different languages than the sample under inspection. In the event that **DIRECT** does not offer enough insight to fix the issue developers can turn to **REINFOREST** for additional context. Once a patch is developed and verified using **ATTUNE** this thesis presents a novel technique called Binary Patch Decomposition (BPD) that allows developers to distribute updated binaries such that software operators can update their software with partial updates that do not break critical functionality even if the entire update introduces backward compatibility issues. These techniques working together form an enhanced software patching procedure that supports difficult patch development circumstances, guarantees the patches fix the bug, and makes sure patches can be deployed without introducing issues in the field.

To illustrate the need for the added complexity consider the following examples.

In the simplest case consider an in-house development team that works to support an internal software product in which the operator, developer, and user are shown in Figure 1.1b may be the same person although far more likely the user is a nontechnical employee who works somewhere else in the company while a single developer or development team both creates and operates
the software. The deployed infrastructure remains under monitoring and when a bug occurs the
technical employee may immediately recreate the bug as opposed to sifting through bug reports and
logs as required by the procedure outlined in 1.1a. The developer can then validate the patch before
deploying the updates for use.

Moving up one level in complexity consider a software service leveraging an open source library
with developers unaffiliated with the open source project that is operated by an external technician
who has a license to deploy the software which is used by external clients unaffiliated with either the
developers or the operator. In the traditional paradigm (Figure 1.1a) developers will struggle even
farther to recreate the exact conditions that caused the problem since they have no access to either
the user or the operator environments. Additionally, the operator in this instance cannot prevent the
developers from issuing patches that break backward compatibility since the two parties are not
intertwined. The proposed update paradigm in Figure 1.1b both facilitates communication between
parties and gives operators recourse in the event of backward compatibility problems.

Enterprise software installations are often even more convoluted as developers use proprietary
third-party libraries that they have no access to or are forced to deal with legacy installations that
lack documentation. In such cases, all the previous challenges remain, but are exasperated by the
fact that developers are forced to work in difficult conditions. That is why the proposed update
paradigm in Figure 1.1b includes development tools to support these difficult circumstances.

First in Chapter 2 we examine the state of the art in related fields that have allowed this work
to take place. In Chapter 3 we present ATTUNE and its patch validation framework along with
an evaluation of its effectiveness on real bugs selected from open-sourced projects. In Chapter
4 we describe DIRECT and evaluate its performance relative to the state of the art in decompiled
identifier renaming. Then in Chapter 5, we introduce REINFOREST and its state of the art code
search capability. The last of the techniques we present is BPD, in Chapter 6, a novel software
distribution framework that demonstrate its effectiveness for real-world software deployments.
Finally in Chapter 7 we discuss some possible future work and present some preliminary studies
showing the feasibility of these concepts.
1.4 Thesis Statement

Incorporating incremental code updates, machine learning-based development tools, and record-replay validation testing enhances software update reliability and robustness. This approach surpasses traditional procedures, which often struggle in the face of legacy code, non-deterministic inputs, and non-backward compatible updates.
Chapter 2: Related Work

2.1 Traditional Procedures

2.1.1 Test generation

iFixR [86] automatically generates candidate patches from bug reports but relies on conventional regression testing to test the patches even though those tests initially failed to detect the bug. Differential unit tests [43] construct unit tests using in-memory program state immediately before invoking the target method, but cannot reproduce bugs not detected by the original developer tests. [89] similarly extracts unit tests from developer execution traces. [171] builds on testing from developer executions but does not pull from live production logs.

KATCH [117] combines symbolic execution with heuristics to generate test cases that cover the patched part of the code, while shadow symbol execution [91] symbolically explores divergences between original and patched versions. Neither leverages execution traces recorded in the user environment, nor fully models system calls, so the generated test cases may not reflect the bug-triggering circumstances. However, symbolic execution enables reaching parts of the program not exercised by the recording, complementing ATTUNE.

Elbaum et al [43] introduced "differential unit tests" generated from the execution traces of developer system tests. Their CR (Carving and Replaying) tool extracts and combines the trace segments that construct the in-memory program state as it was just before invoking the target Java method, which then serves as a unit test. CR also complements ATTUNE, since its system tests would likely exercise the program more broadly than available user execution traces. Since CR does not leverage user execution traces and its system traces support only in-memory events, its tests may not reflect known bug-triggering user environments. Other work similarly extracts unit tests from developer execution traces, e.g., [89], with analogous advantages and disadvantages. CR
does not attempt to continue to replay through the execution of the method under test, the method’s return to its caller in the full system execution, and beyond. In contrast, ATTUNE’s ad hoc tests are generated from system recordings made in user rather than developer environments, and the primary goal is indeed to continue replay of the full system through the execution of every changed function until its clear the bug no longer manifests - a more challenging problem. ATTUNE requires faithful replay as a baseline, emulating interactions with files, databases, and other resources in the user environment, whereas CR replays only in-memory program state.

2.1.2 Dynamic debugging techniques - MVE and Fuzzing

Multi-Version Execution (MVE) provides an alternative approach to user validation. In MVE, the patched and original versions run simultaneously on production user workloads, adding runtime overhead but enabling immediate detection of undesirable divergences [66, 143]. These papers also do not address switchover, in how a newly patched version is introduced into an already running MVE user environment. Unlike MVE running different code versions, as in ATTUNE, LDX [94] runs two instances of the same code to infer causality between events. The slave changes one event from the master execution to find divergent impacts on later events, orthogonal to our work.

Fuzzing seeks inputs that induce crashes and other problems [144]. Other approaches also strive to induce bad behaviors, e.g., [12, 186]. [164] builds on EvoSuite’s search-based testing [50] to reproduce crashes. Symbolic execution [20] and other approaches generate test suites to achieve coverage goals. There is a rich literature concerned with generating inputs intended to trigger or reproduce bugs. Generally, the same generated tests could be applied to multiple program versions - unless those tests are "flaky". There has also been much work towards making tests repeatable, which is sometimes difficult even in the developer environment on the same system build [98]. These kinds of tools, as well as conventional regression testing, are complementary to ATTUNE.
2.1.3 Live Instrumentation and Debugging support

Parallel retro-logging [148] allows developers to change their logging instrumentation so previous executions produce augmented logs, but the program is not modified. Network-level traffic cloning tools relay or replay only network inputs for service-oriented and microservices architectures. Parikshan [9] feeds cloned network traffic to a sandboxed component of a service-oriented architecture, for debugging or testing patches of that component, but the sandboxed execution is not faithful for non-network sources of non-determinism.

2.1.4 Looking Forward

Most of the works cited above tend to fall into traditional computing techniques and system/design architectures. However, as AI continues to impact the way we create and think about software it will enhance the above technologies as well. While completely separate and apart from the technologies listed above as it deals mostly with source code instead of systems-level programming, such functionality would greatly enhance the effectiveness of the tools above. Most of the tools above are designed to offer more context and operational alternatives to developers, but instead, these same tools could offer context and operational alternatives to AI agents to identify problems and make fixes.

Consider work like TOGA [39] which uses transformer-based architectures to infer the intended functionality and develops a test of the intended functionality instead of the implemented functionality. Combined with some of the execution tools listed above the test case generated no longer requires developer oversight, and instead becomes a stand-alone unit test.

Since most current transformer work is based on LLMs, we tend to integrate AI at the source code or other human interpretable level, but we can go in a different direction. Consider WeReplay [46] a record replay framework that leverages information in the GUI of a mobile application to replay interactions. They use AI in this case to infer machine state from the human interface. Considering the amount of data collected from recording traces and the overhead associated with those collections an AI-enhanced log that infers relevant pieces of machine state would not be
completely deterministic but would have significantly less overhead and deployment requirements. Reflecting on something like REPT (mentioned above) instead of using deterministic logic to recreate the events of the past there is no reason AI could not be used to do the same thing with even less context.

While there are of course the engineering and research challenges that remain before bringing these visions to life the applications of AI in similar fields make this evolution increasingly likely. What remains an open question however is the trade-off with reliability. By definition, AI uses fuzzy logic instead of the traditional techniques outlined above that rely on deterministic operations. When concerned with security matters, reliability is of increased importance, and as a community we have not yet determined if AI-produced content is fit for security-critical contexts.

### 2.2 Record/Replay Validation

2.2.1 Recording

In the context of discussion around this thesis, some questions have arisen around recording, the trade-offs associated with such recording, and the effectiveness/limitations of various types of telemetry. Therefore even though many techniques are included in the previous sections and the contributions of this thesis make no efforts to advance recording technology an additional discussion around this subject is included here.

First, let's consider the various types of bugs that we try to capture using record replay analysis.

Table 2.1 lists bug types going from least to most complex from top to bottom. In general, the techniques that can accommodate the higher levels of complexity also extend to the lower levels of complexity (although not a strict rule). Table 2.2 lists the various telemetry strategies going
### Table 2.2: Telemetry Sources

<table>
<thead>
<tr>
<th>Telemetry Sources</th>
<th>Description</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input-based</td>
<td>Records only inputs like keystrokes or data payloads</td>
<td>[145, 6]</td>
</tr>
<tr>
<td>API-level</td>
<td>Focuses on API-calls and network traffic for recreation</td>
<td>[199, 109, 122]</td>
</tr>
<tr>
<td>System calls</td>
<td>Captures all system calls &amp; arguments made to the OS</td>
<td>[48, 136]</td>
</tr>
<tr>
<td>Memory Accesses</td>
<td>Software behavior related to reading/writing memory</td>
<td>[200, 136]</td>
</tr>
<tr>
<td>Thread Scheduling</td>
<td>Timing of threads relative to one another</td>
<td>[147, 136]</td>
</tr>
</tbody>
</table>

Figure 2.2: Telemetry Sources

from least to most overhead (recording additional sources adds costs in performance or complexity).

Note: this is not a completely strict taxonomy as there is some overlap between techniques.

Design decisions around what type of recording system to use depend largely on the bugs you must reproduce. In the event that user interaction is sufficient to replicate the bug, then lightweight systems that only monitor user interactions are sufficient. This is especially useful in application debugging where the software in question is user-facing. For other software that deals primarily with behavior driven by events, API call monitoring captures the required context to recreate the buggy conditions. This has gained popularity in cloud/micro-service environments as network communications are the only common denominator in disparate execution environments [122].

Moving towards heavyweight solutions, like those used to generate the verbose logs, they must capture all system activity as well as activity initiated by the user. This includes all system calls and all operating system events. Perhaps the most difficult bugs to recreate are shared resource bugs related to thread interleaving etc. In these cases a full system replay is required and must include timing information. In the simplest case collecting user input adds minimal overhead to the underlying software’s performance and complexity. Collecting a system’s API calls to external servers adds minimal complexity and overhead as the data in both cases is readily available and easy to copy. Furthermore, API calls and collecting user input comprise only a small portion of most software activity, so most software execution remains uninterrupted. In order to collect more detailed system information like system calls, thread scheduling, and memory access patterns most software functions must be interrupted and thus the recording mechanism adds significant overhead. Further, since the logs record everything the software is doing, the resulting logs are significantly larger and more complex to deal with. Some more recent work with really low recording overhead
investigates actually changing hardware performance based on memory access patterns [200] for performance gains. However, they achieve such good performance only by monitoring specific structures. While especially useful in their domain, it would be interesting if such concepts could be extended to a more general replay.

Successfully recreating thread scheduling bugs remains one of the trickiest problems in practice due to the large scale of modern cloud systems. Solutions like [21] use some sort of log to record memory accesses by threads and identify overlaps. Such overlaps become the most important parts of the replay. Still tracking such a state imposes impractical overhead on the system. [136] relies on a hardware performance counter for thread timing, and as stated elsewhere in this thesis imposes unrealistic overhead on the runtime. Capturing thread interleaving at scale in practice remains an open problem in many ways, but we have some tentative solutions.

Table 2.3 outlines the telemetry sources that must generally be collected to recreate the buggy behavior of each category.

2.2.2 Record Replay and Mutable Replay

Kravets and Tsafrir [88] proposed "mutable replay," a hypothetical design to construct a new execution trace for a modified program from an execution trace of a previous program version that, as in ATTUNE, demonstrates a bug. Mutable replay was later implemented by Viennot et al. in Dora [177], building on the Scribe record-replay system [96]. Dora leveraged checkpoint/restart [95] in a backtracking search algorithm that sought to minimize adds/deletes to the original execution trace. Although successful on many bug-fix examples in the sense that execution continued through the modified code, the minimal-distance execution trace is not necessarily the same as would have
occurred had the modified code been running in the user environment, which is what ATTUNE aims. The underlying Scribe record-replay required a shared file system (copy on write) between the user and developer environments and a special Linux kernel module that intercepted and controlled system calls and other non-deterministic kernel events within both user and developer environments, which are impractical for most post-deployment scenarios, whereas ATTUNE runs without privileges in user-space with no changes to the operating system and no sharing between user and developer environments other than user-submitted execution traces.

There are numerous other record-replay tools in the literature, e.g., [202, 106, 107, 159, 52, 123]. These tools reproduce executions, but none tests patched versions. Some versions of gdb build-in recording and replaying debugging sessions, as does Microsoft’s IntelliTrace [123]. These tools reproduce executions for a program version and cannot test patched versions. Many record-replay tools focus on reproducing concurrency bugs, e.g., [21, 70, 152], outside the scope of this thesis. While ATTUNE supports ad hoc test generation for multi-threaded programs, our prototype built on rr cannot generate tests for patches aimed specifically at concurrency bugs due to how the rr implements multi-threading (it simulates multiple threads within a single thread). Much research focuses on reducing recording overhead, e.g., [141, 68, 77]. Cui et al [25] explains that "high-fidelity program tracing is not affordable in deployed systems", so their REPT tool combines hardware tracing and binary analysis to reconstruct execution traces, which can then be replayed with the same program version. Castor [118] records multi-core applications by leveraging hardware-optimized logging, transactional memory, and a custom compiler. It can replay slightly modified binaries that do not impact the program state. Some research trades off faithful replay guarantees, e.g., to better address long-lived latent bugs for time-travel debugging [127].

Although some papers about record-replay systems refer to capture-replay, e.g., [166], record-replay as discussed in this thesis is different from most capture-playback tools. These record or script user actions to repeat for GUI compatibility testing across multiple operating system versions, browsers, or devices [124, 158, 8, 132, 81]. Capture-playback is conceptually similar to ad hoc test generation. Still, these tools focus on externally visible behavior and are not intended for faithful
bug reproduction or testing security patches.

2.3 ML/AI Based Development Tools

2.3.1 Variable Name Recovery and Type Inference

DIRE [97], compared to in the evaluation, performs the same task as DIRECT but uses traditional LSTMs combined with GGNNs. DIRECT uses DIRE’s tokenizer as is, our innovations replace DIRE’s bidirectional LSTM with our task-specific transformer architecture. Before DIRE, Debin [62] represented the prior state of the art using decision tree-based modeling.

**Type Inference:** Debin also attempted to recover type information – which is a different problem. Debin continues to serve as the non-neural-based state-of-the-art for this task. Typilus [4] is a new GGNN-based approach for type inference. that represents the current state of the art.

**Function Name Recovery:** An orthogonal decompilation problem is function name recovery. Function names are usually left in executables’ metadata, by default, but in malware, these symbols are probably stripped. Recent work by Artuso et al. [10] has shown transformers are highly applicable to this task and the pre-training/fine-tuning paradigm has a place in code analysis. Still, they limit their experiments to function names. Other work like David et al. Further work on this like CFG2VEC [197] and SymLM [74] deal with function name recovery leveraging transformer-based architectures demonstrating that proper feature extraction and code modeling can have a significant impact on the generality and success of the approaches. [35] uses LSTM architectures to encode API call sequences as function profiles and learn the function names commonly associated with those call sequences. More recent investigations like [41] show that choosing training data with diverse variable names can change performance significantly, but training data investigations fall outside the scope of this thesis.

2.3.2 Transformers for Filling-in Blanks:

Filling in blanks in an input sequence necessitates a model that can capture bidirectional context. BERT’s pre-training objective [36] solves this problem by reconstructing random masked
tokens. SpanBERT [76] focuses on contiguous spans of masked tokens with a modified pre-training objective. These methods require the location and length of each blank to be known in advance, but Insertion Transformers [165] solve for variable-length blanks without explicitly controlling insertion.

Blank Language Models [160] solve for fixed blanks with variable length with a special blank character that the model can predict and feedback in a loop. Another architecture that solves the same problem is BART [102]. Like us, BART uses a BERT encoder and a left-right decoder to perform arbitrary transformations on the input. However, neither of these approaches can be directly applied to variable renaming without modification to guarantee that multiple blanks have the same prediction.

Before transformer-based models like BERT, ELMo [146] attempted to capture the token context in the embedding procedure. ELMo was built using bidirectional LSTMs. After BERT’s creation, a series of BERT variants were published. One such variant is Facebook’s RoBERTa [108]. RoBERTa is essentially an optimized BERT model. The next-sentence objective is removed from pretraining, and various hyper-parameters like batch sizes, training times, and data sets are adjusted.

Facebook also released XLNet [192] to deal with longer sequences and reduce memory consumption. Google released their own optimized version of BERT called ALBERT [99]. BERT’s enormous model takes a long time to train so ALBERT presents two parameter reduction techniques to minimize training time and memory overhead. Using ALBERT instead of BERT for variable name recovery would be an interesting study for future work. Other variants are not architecture but simply pretraining the same architecture on different corpora. One such project is CodeBERT from Microsoft Research [47]. CodeBERT is pre-trained on source code functions and their corresponding documentation. It was evaluated on the tasks of code search and automated documentation. While a similar application of BERT, it has no context of binary or structural information in the decoder or encoder processes.
2.3.3 Decompilers

There are two decompilers used in practice. One is Hex-Rays [64], from which DIRECT’s training set was built, and the other is the open-source Ghidra platform [53], which both fail to make meaningful efforts at reconstructing variable names without debugging information. A research compiler DREAM++ [190] functions signature heuristics to generate meaningful variable names but does not apply ML models.

Recent works like [149] and [40] have also investigated the difficulties of applying ML models from other disciplines directly to software engineering tasks.

2.3.4 Future Directions

Most of the work in this area has consisted of applying LLMs to source code, or some extension like applying LLMs to human-legible code extracted from binaries. This is only logical as LLM’s are trained for human legible language. However, investigation into tasks where context is strictly at lower level abstractions like data type recovery (mentioned earlier) remains sparse as datasets do not exist and are hard to create. Some attempts to try and tackle these types of problems like Callee [204] (which identifies indirect jump targets from binary) rely on transfer learning.

Regarding the problem of code search, research endeavors on traditional source code to source code search in the same language have become exceedingly rare as the field progresses into more complex problems like cross-language code search evaluated in this paper. One area of code search that has become more popular is cross-architecture code search like that described in VulHawk [113]. They demonstrate that cross-language code search is effective even at the assembly language level. Reconsidering some of the traditional code search problems at a lower level of abstraction remains an exciting area of research, but it is quickly closing. Further, as language models become the predominant form of AI, dynamic analysis becomes less appealing since the overhead associated with dynamic properties is no longer worth the trouble. As we discuss later in this thesis, we show that it is possible to infer dynamic properties from static chunks of code. We only examine inputs and outputs, but some other dynamic analysis generated from static inputs remains an open area
of research. For instance, given the performance of a given implementation in specific scenarios, can we predict how a similar implementation would perform, or even given the problems that arise with particular software architectures at runtime, can we evaluate how parallel architectures can accomplish the same tasks with less downside?

Looking forward, it seems clear that AI has displaced other methods as the primary approach for many SE-related tasks, but the best way to use AI remains a question. Whether it is best to raise the context to human legible language (like source code) and run language models at that level or try to operate directly on the binary remains an open question. Further, it remains a subject of investigation as to whether we should try to tackle these problems directly by creating new datasets and specially trained models we should rely on foundational models which have been trained on significantly larger datasets outside of the domain, or if perhaps transfer learning from similar available datasets to the domain in question are the best approach.

Looking further forward, it seems highly likely that humans will not just consume the outputs of these analyses but will instead be consumed by autonomous agents with directives to patch, create, monitor, and refactor code. Such agents and their roles/capabilities remain an open question.

2.4 Incremental Code Updates/Patching

2.4.1 Hot Patching and Binary Rewriting

Binary rewriting has been used for many reasons including implementing defenses, automatic program repair, hot patching, and optimization. Hot patching is an interesting example since it requires conserving the dynamic program state at the time the repair is applied similar to binary quilting. Katana [150] has highly sophisticated mechanisms for handling this problem, many of which would augment our current quilting procedure, but relies on trampolines to apply the patches which could incur significant overhead the same as [73]. ATTUNE avoids the need for trampolines by recompiling the whole binary, thus avoiding the overhead associated with trampolines. Other binary rewriting mechanisms like Zipr [60, 65] raise the binary to a higher level IR which allows for increased efficiency in the reassembly process similar to Egalito [185] but have demonstrated
generic binary level defense transformations instead of semantically complex bug specific patching. [101] outlines the difficulties in dealing with legacy systems and the usefulness of binary rewriting in those situations.

As binary rewriting becomes more accessible and incurs less performance tradeoffs it seems likely that this kind of work will be more practical. Still, there remain many situations in which binary rewriting is not possible as the compiler has removed too much source-level information. For these techniques to become ubiquitous significant scientific advancement is required.
Chapter 3: Ad Hoc Testing
3.1 Introduction

Developer testing may not represent how software is used in the field [178]. The existence of the Common Vulnerabilities and Exposures (CVE) list for security vulnerabilities (http://cve.mitre.org) [170] and mundane user bug reports [22, 15] demonstrate that not all bugs are User bug reports [86, 22] and vulnerability reports [34, 58] are populated primarily with bugs that were not discovered by developer tests. But reproduction steps included in bug reports are often insufficient to reproduce the bug [23]. Still, disclosures and other bug reports often include evidence of the bug, such as memory dumps, stack traces, system logs, error messages, screenshots, etc. Thus, when a security vulnerability or other critical bug is not detected by developer testing before deployment, but reported by users, developers need to construct a new test that reproduces the bug in the original code version and verifies the absence of the bug in the patched code. Furthermore, the same test (or additional new tests) must verify the presence or absence of the bug in candidate patches while still passing the developers’ original tests. Although minimizing the time from bug discovery to patch release is of the essence, users are wary of rushed patches, since they may break mission-critical functionality [195]. In contrast, user validation remains difficult and time-consuming.

This chapter presents a novel approach for rapidly generating tests that reproduce post deployment bugs, support debugging, and verify that candidate patches do not exhibit buggy behavior. We build on state-of-the-art record-replay technology that records execution traces in the user environment, reproduces post deployment bugs, and supports debugging. Still, state-of-the-art record-replay tools do not support testing candidate patches. Since bug-fix patches may introduce other errors not detected by developer tests, breaking users’ mission-critical functionality [195], our approach also supports user validation of released patches.

The gist of our patch-testing solution is illustrated abstractly in Figure 3.1. Some always-on lightweight recording mechanism records production applications. When a bug is discovered, the lightweight recording is replayed and re-recorded offline to augment with additional context necessary for replaying in a different environment. The verbose log is shipped to the developer, who
Figure 3.1: Ad hoc Test Generation Concept

writes and verifies prospective patches. Finally, the new release is accompanied by patch-specific metadata enabling users to validate the patch.

We refer to this concept as *ad hoc test generation* because the generated test emulates whatever user context manifested the bug. We emphasize that ad hoc test generation is intended only for urgent time-crunch situations when there are no existing developer tests that detect the bug and careful planning and design of new developer tests would take too long. Ad hoc test generation is feasible when execution trace divergence is small, analogous to Tucek et al.’s “delta execution” [172], whose large-scale study of patch size found that security and other patches solely to fix bugs tend to be modest in size and scope, rarely changing core program semantics, shared memory layout or process/thread layout.

The premise of record-replay technology is that behaviors manifest in the user environment that cannot be reproduced by simply running the program with known inputs in the developer environment. If there are such known inputs, ad hoc testing is easy – just run the program with those inputs. This chapter addresses more complicated scenarios.

We have developed a novel ad hoc test generation tool, ATTUNE (Ad hoc Test generation ThruUgh biNary rEwriting). Instead of requiring developers to build doubles, mocks, or other test scaffolding to fake the user environment for its tests, ATTUNE builds on existing record-replay tools:
It emulates the original execution context, including external inputs, environment variables, the results of system calls, network connections, and the accessed portions of the file system, databases, and other local resources as they were at the time of the exploit or bug manifestation. Unlike existing record-replay tools, ATTUNE leverages binary rewriting [182] to modify the original executable at load-time to insert the patched functions from the modified executable, and then interprets the recorded log to manipulate the test emulation as it executes the patched functions. Inserting the patched functions into the original binary results in an execution that perfectly matches the recorded log until divergence when the first patched function is reached. Continuing the execution beyond this point enables the developer to assess whether a candidate patch indeed fixes the bug.

ATTUNE is based on two key insights: The first is that the symbol tables in Linux ELF files provide reference points between original and patched versions. Thus, each patched function can replace the original function and access the same global variables and strings. Our second key insight is the recorded log of the original execution trace does not need to be replayed verbatim in order. Instead, events in the log can be skipped or swapped, and new events can be derived on the fly from those in the log, to match the patch.

ATTUNE follows the workflow illustrated in Figure 3.1 to generate an ad hoc test for candidate patches faithful to the execution trace recorded in the user environment. Since ATTUNE requires detailed traces that would impose too much time and space overhead for always-on recording in user environments, we envision that always-on recording is performed by a lightweight record-replay mechanism like Castor [118]. Then, when user observation, analysis, monitoring, etc. determines that a lightweight trace manifests a security vulnerability or other bug, then that trace is augmented offline by ATTUNE’s verbose recorder.

Our ATTUNE prototype builds on Mozilla’s rr open-source record-replay tool [138, 137, 129] as the verbose recorder. ATTUNE like rr runs without privileges in user space, with conventional hardware, operating system, compiler, libraries, build processes etc., and no changes to the application. This starkly contrasts other test generation techniques like symbolic or concolic execution. ATTUNE’s binary rewriting modifies the application executable only at load-time, i.e., in memory.
not the executable file(s). While the technical details of our binary rewriting mechanisms are specific to our modification of rr’s replayer, ad hoc test generation is not, and in principle ATTUNE prototypes could be built on any record-replay technology that supports sufficiently detailed execution traces.

Our ad hoc test generation scheme is complementary to developer tests. The existing developer test suite should run on any prospective patches, to check that the patch does not break behavior previously tested by developers. Since regression testing is likely to execute the same paths in the old and the new versions of the code, it might be augmented by shadow symbolic execution, which generates tests specifically exercising differences between patched and previous versions [91]. ATTUNE emulates only the specific, probably unusual circumstances under which the bug manifested in the field, which is not addressed by shadow symbolic execution. Our premise is that none of the existing developer tests anticipated those circumstances.

To manually write a new test in response to the security vulnerability, the developer would need to write setup code that constructs the program state that the buggy function depends on or affects (or each buggy function, if more than one contributed to the vulnerability) equivalent to the corresponding portion of the program state as it existed in the user environment just before entering the buggy function(s). The developer would also need to devise mocks and test doubles or fakes for relevant user environment resources (external inputs, communicating processes, files, etc.). The developer might write a simpler test that executes much faster than ATTUNE, whose ad hoc tests take about five seconds to generate and run. But five seconds is almost certainly less time than a developer would take to manually write the new test.

We explain our requirements for verbose execution traces and the technical details of our binary rewriting techniques in Section 3.2. Our evaluation in Section 3.3 describes how a developer would use ATTUNE to test candidate patches for a variety of bugs from well-known open-source projects. Section 3.3 also gives an example where the user records their own workload with the original program and replays with the modified program to convince themselves that the bug has been fixed and the patch does not break other behavior. Finally, our evaluation compares the time and space costs of using KLEE [84] to generate a test that covers the patched part of the code, given that the
bug has already been found and its location is known.

The contributions of this work are:

• An approach to leveraging record-replay technology and binary rewriting to generate ad hoc test cases to exercise candidate patches as if they had been executing in the user context, instead of the previous buggy version, when the bad behavior was originally recorded.

• A technique for adding developer environment metadata to patch releases, enabling users to validate patched versions with their own workloads by (re-)recording with the old version and replaying with the new version.

• An open-source prototype implementation, portable across Linux distributions running on x86-64, available at https://github.com/Programming-Systems-Lab/ATTUNE.
3.2 Architecture and Design

Our ad hoc test generation workflow has four main components: recording, static preprocessing, loadtime quilting and the runtime replay decisions. Recording and the two preparation stages are shown in Figure 3.2, with runtime depicted in Figure 3.10. Both preparation stages leverage the open-source Egalito recompilation framework [185].

3.2.1 Recording

We assume production recording with the user’s choice of lightweight tool and, when warranted by some external mechanism that detects an error or exploits, offline replaying that tool’s recording while re-recording with rr’s recorder as in Figure 3.1. Instead of rr, any other recording engine that constructs sufficiently verbose traces would suffice, but we do not know of any actively-supported open-source alternatives. Specifically, the trace must provide the details needed for ATTUNE to
recreate the successive register contents and memory layouts leading up to when the bug manifested. Thus the recorded sequence of events must include register values before and after system calls, files that are mmapped into memory, and points at which thread interleaving and signal delivery occur during execution.

3.2.2 Static Preprocessing

**Source Code and Binary Preprocessing.** Figure 3.3 shows an abbreviated example patchfile from a libpng bug-fix [154]. Patchfiles document which files changed, which function in the file changed, and which lines within that function were inserted and deleted. Patchfiles are created with a standard format so we are not limited to a single diff implementation.

**Dwarf Information & Symbol Table.** Patch files don’t provide any information about the resulting binary. Since the recorded trace relies on binary/OS level information (register values, pointers, file descriptors, thread IDs, etc.), we need to translate from source changes to binary changes.

```
182: 0000000000003fe0 56 FUNC
    GLOBAL DEFAULT 1 png_check_chunk_name
183: 00000000000004020 221 FUNC
    GLOBAL DEFAULT 1 png_check_chunk_length
184: 00000000000004100 172 FUNC
    GLOBAL DEFAULT 1 png_read_chunk_header
```

**Figure 3.4:** libpng-1 Symbol Table Entries

Two mechanisms enable this translation: The first is the symbol table standard in all ELF files
and the 2nd is DWARF information. The key insight is that the **symbols act as a point of reference between the old and the modified binaries**. They remain unchanged even if their addresses and references change. After processing the patch file we use the symbol tables to find the locations of functions and global variables, and we use DWARF information for finding changed lines and identifying source files. These two sources combined contain all the information in the source level diff at the binary level. Refer to Figures 3.4 and 3.5 for concrete examples.

Most real-world builds create multiple binaries and associated libraries, so it may be unclear which binary contains the associated change. To generalize to sophisticated build processes ATTUNE uses DWARF information to search through all re-compiled binaries to find the modified file.
3.2.3 Load Time Quilting

**Pre-Load Steps for Quilting.** Once the function and line addresses have been resolved, and a prospective patched binary has been compiled, we can generate our test code. For the newly compiled patched code to remain a viable test case, it must maintain the binary context of the original code. While most binary context remains unchanged, code pointers and data pointers that point somewhere inside the modified functions or that point from the modified functions to any location outside the modified binary must be updated accordingly. To create the most accurate test we point to the original binary context wherever possible. To fully integrate the patched code with the recording, references to shared libraries must point to where the shared libraries were loaded in the recording, references to places in the modified section of the code must point to the appropriate place in the patched code, and references to unmodified contents of the patched binary must point to the appropriate place in the original binary as in Figure 3.6.

To prepare for load time quilting resolution (explained shortly), static reference identification must occur for bookkeeping purposes. The patched function is scanned for all symbol references that must be resolved to integrate with the recorded context. Some references like references to locations within the modified function (*e.g.*, jump and conditional jump instructions) can remain unaltered in position-independent code. So after all references are accounted for, they are trimmed to the subset of references that need to be changed during the quilting procedure. This includes references to strings, shared library functions, functions that only exist in the original or the modified binary, functions that exist in both, procedure linkage table (PLT) entries, and global variables. Since symbols are the points of reference between original and patched binaries because recompilation renders addresses meaningless, references to be resolved are defined as a symbol and an offset from that symbol.

**Loading Replication & Custom Loading.** In modern Linux systems, the system loader parses the executable’s header, loads it into memory, and dynamic links. Since shared libraries are not always loaded at the same positions, references related to the global offset table (GOT) and procedure linkage table (PLT) are resolved after loading is complete. So even though ATTUNE
knows pre-load which references need resolution, it can’t actually resolve those references until load time. To preserve the integrity of the replay, all required shared libraries, executables, and system libraries must be loaded into the recorded memory locations. Shared libraries and executables required for replay are included in the trace, and non-recorded libraries loaded during replay are limited to the system loader, which is required at the start of any process.

To replicate the recorded loading activity, ATTUNE begins by loading a small entry point program (replay hook), which hijacks execution from the system loader and begins the replay process. As mentioned earlier, some references in the patched code can’t be resolved until the original code is loaded into memory, so initially loading replicates exactly what was recorded. Once the original segments are loaded into memory and GOT/PLT relocations are completed, ATTUNE resolves the remaining references in the patched code (described below).

Finally, ATTUNE’s loader loads the quilted code after finding an appropriate place to put it. Note quilting has to be repeated on every replay, and the files containing the original and patched executables are not modified. The loader searches the address space for the lowest slot large enough to accommodate all of the patched code, then loads the patch following the Linux loading conventions. Figure 3.6 depicts the address space when loading has been completed, and Algorithm 1 outlines the loading procedure.

**Address Translation Procedure.** A summary of the procedure to translate pointers from the context of the modified binary to the original binary is given in Figure 3.7, and consists of

---

**Figure 3.7:** Pointer Translation Procedure

![Diagram of Pointer Translation Procedure](image-url)
Linking function: png_check_chunk_length
   in module pngutil
Updating Instruction Reference
   from [0x1b214] to [0xaa60]

//identifying reference point
Target Symbol: png_chunk_error
Offset From Symbol: 0
Symbol Location in original binary:
   0xaa60

//target address in the original binary
Target Address: 0xaa60
...
//patch references string
Resolving string reference at: 0x1b2cd
Resolving offset ...
   for "chunk data is too large"
//identified string in original binary
Found string: "chunk data is too large"
   at 0x320e
... module pngutil code found at 0x000000
... module pngutil data found at 0x200000
... generating quilted code

Figure 3.8: libpng-1 abbreviated linking example
Algorithm 1: Custom Loading Algorithm

<table>
<thead>
<tr>
<th>Result:</th>
<th>Load patched code into the address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>code_seg_size = 0;</td>
<td></td>
</tr>
<tr>
<td>char* code_buf;</td>
<td></td>
</tr>
<tr>
<td>for func in mod_funcs do</td>
<td></td>
</tr>
<tr>
<td>code_seg_size += func.size;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>for segment in addr_space do</td>
<td></td>
</tr>
<tr>
<td>space = next_segment.start - segment.end;</td>
<td></td>
</tr>
<tr>
<td>if space &gt; code_seg_size then</td>
<td></td>
</tr>
<tr>
<td>start = segment.end;</td>
<td></td>
</tr>
<tr>
<td>for func in mod_funcs do</td>
<td></td>
</tr>
<tr>
<td>patched_code = func.gen_code();</td>
<td></td>
</tr>
<tr>
<td>code_buf += patched_code;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

both pre-load and load-time actions. The process starts from the address of the modified function as determined by the patchfile and DWARF processing. The modified function is scanned for references. When a reference is identified, if the pointer is affected by the quilting process then ATTUNE’s translation procedure corrects the pointer.

The log messages in Figure 3.8 explain the process in detail: An instruction in the patched binary at 0x1b214 points to 0xaa60. In order to update the instruction to point to the same position in the original binary we need to identify the correct symbol and offset in the original. First, we convert the target address 0xaa60 into a symbol and offset in the patched binary. Since this instruction is just calling a function, the target symbol is the function name and the target offset is 0. Then ATTUNE searches the original binary for the same symbol and offset, and in this case, the function was generated at the same address in the original binary. Resolving string references, global variable references, and PLT references require slightly different procedures and are described below. Finally, the patched code is generated with instructions pointing to the correct locations at runtime.

**PIC Code, PLT Entries & Trampolines.** Position-independent code compilation has become
the standard for security and efficiency reasons so that modern binaries can be loaded anywhere in the address space. As a result, the locations of external functions and symbols are unknown until those symbols exist in the address space. Since most library functions aren’t called, they aren’t all resolved at load time and are resolved only after they are called. The procedure linkage table (PLT) acts as a table of tiny functions that perform a function lookup and trampoline to where the code for external functions is defined.

Unfortunately we can’t rely on a PLT because the system loader that performs the runtime function resolution doesn’t know about ATTUNE’s special memory configuration. Two key differences let us implement static trampolines instead of relying on the traditional PLT mechanism. 1) We only need to resolve the PLT entries that are referenced by the modified code, which comprise a small fraction of the overall PLT, and 2) we can resolve these beforehand without relying on the PLT’s lazy loading mechanism because the shared libraries have already been loaded by the time this code is injected. The x86_64 architecture only allows call instructions with a 32-bit offset, but we need to call functions across the 64-bit address space to reference shared library functions. To accomplish this we transform calls to PLT entries into a move instruction that loads an address into a register, and then a call instruction to the address in the register, as shown in Figure 3.9.

**Resolving String & Data Sections.** The patched code may also reference data section variables like global data and strings. The patched code must reference the old code where possible and the patched code where required. Identical symbols and strings function as points of reference between the modified and the original binary.

These translations are similar to Figure 3.7, with a few minor differences: String tables don’t have an associated symbol table. The modified code references the string directly, but to look up the location of a specific string in the original, we have to iterate through all of the read-only data. If
the string exists in the original binary, we point at it, otherwise ATTUNE points to the appropriate location in the new data section. Note the binary normally accesses data through a global offset table entry, but cannot use it here because the global offset table was compiled for the modified code. Instead, ATTUNE transforms the binary to point to the data directly, since it knows where the data has been loaded.

### 3.2.4 Runtime Replay Decisions

The runtime architecture is shown in Figure 3.10. At runtime, we continue to leverage developer environment information to aid ATTUNE’s decision-making since we know exactly which functions have been modified and perform a strict replay until a modified function is called. We break at that point and move to the patched code, where we use information about added or deleted lines to inform decision-making.

For any non-deterministic event that takes place during replay, we must decide whether to use a corresponding event recorded in the log or to actually submit the event for operation by the
kernel(execute live as would be required if the inserted code makes a new system call). We emulate kernel state and kernel events whenever possible, and only ask the kernel to perform the replaying action when necessary, following the greedy approach shown by the pseudocode in Algorithm 2. It should be noted that system calls that depend on process states, like malloc, and mmap, don’t require emulation since this state is actually recreated during replay. All file operations performed during replay are based on information available from the recorded trace, essentially recreating how the program would have acted at the time of the bug except now (for successful patches) without the bug. If there is no further information available, the emulation ends.

**System Calls.** The simplest event types to replay are system calls that don’t involve file IO. We can reuse results from the log if the parameters for the syscall match what is in the log. It won’t match the log exactly since the log contains checks for all registers including the instruction pointer which is obviously different, but we relax these checks once replay has diverged to only check registers containing syscall parameters.

**File IO.** System calls involving file, network or device IO are harder to replay since they require a specific kernel state. We have to recreate the file state so we track `open`, `close`, `stat`, `read`, `write`, and `seek` operations for all file descriptors during replay. At the point the replay diverges we have a partial view of the file system. Of course, we can’t recreate any data that doesn’t exist, but if a file operation can’t be satisfied during replay we can look forward in the recorded trace to see if we have enough information to satisfy the operation. If we do then we emulate it, and unfortunately, if we don’t we have to die. Another approach would be to supply random bytes, but we feel this wouldn’t accurately reflect a realistic state if the full file system were available.

**Signal Delivery.** If the emulation engine intercepts a signal, we need to decide if that signal should be delivered to the replaying process. Our normal replay mechanism based on rr’s replay mechanism determines when to deliver signals based on the value of the retired conditional branches (RCB) performance counter standard in Intel chips. We check if we are in an inserted line for recorded signals based on signal type. If we are then we deliver the signal and assume it’s created by the patch (e.g., a segfault from an incorrect memory reference in the patch). However, if a recorded
signal is delivered and we are not currently in the inserted section of the code we can do our best to estimate at what RCB count it should be delivered by taking the target RCB count and adding the number of RCBs caused by inserted lines. While this isn’t perfect, it allows for a rough idea of when the signal should be delivered. If an unrecorded signal fires we allow that signal to be delivered without interference since there is no recorded timing information to guide delivery.

**Algorithm 2: Runtime Replay Algorithm**

- **Input**: `e`: an event that stops replay
- **Output**: The next event to replay

```c
Function getResult(e):
    if !diverged then
        return next_recorded_result;
    end
    if is_syscall_without_file_io && exists_unused_in_log then
        return recorded_result;
    end
    if is_syscall_with_file_io && supported_operation exists_unused_in_log then
        return recorded_result;
    end
    if is_signal && signal_is_recorded then
        if current_pos == inserted_code then
            return nullptr; // execute live
        end
        return DELAY; // delay until RCB count
    end
    return nullptr; // execute live
return
```
3.3 Evaluation

The ATTUNE implementation only works on Intel chips with the x86_64 instruction set since we leverage the expanded instruction set’s modern instructions to perform binary transformations. We evaluated ATTUNE on a Dell OptiPlex 7040 with Intel core i7-6700 CPU at 3.4GHz with 32GB memory, running Ubuntu 18.04 64bit, using gcc/g++ version 7.4.0 and python 3.4.7. ATTUNE is built using CMake version 3.10.2 and Make version 4.1. For comparisons with KLEE [19], we used KLEE version 2.1 [85] (the most recent version as of this writing) compiled with LLVM 9.0.1. To keep our evaluation current and correct we limited our study to bugs that have occurred since 2016 and have since been patched. Where possible we made an effort to focus on security-related bugs since these are the bugs most likely to require immediate attention without taking time to develop formal test cases. Since ATTUNE’s envisioned use primarily concerns server-based applications we limited ourselves primarily to databases and libraries, but ATTUNE could be extended to work with GUIs as well.

Since we want to evaluate ATTUNE on an unbiased selection of patches for both security vulnerabilities (CVEs) and other kinds of bugs, and know of no benchmark that provides user environment execution traces or scripts to set up the user context for recording traces, we recruited (for one semester of academic credit) an independent challenge team of three graduate students who were not involved in developing ATTUNE nor versed in how it works. They were tasked to identify a diverse collection of around 20 bugs in widely used C/Linux programs. The bugs had to have been patched during 2016–2019 and the students had to construct user contexts that demonstrated the buggy behavior. For example, to recreate the circumstances leading up to the redis-1 bug, first one needs to run the server with a specific configuration, connect to the server in MONITOR mode, and then send a specific byte stream. Note the team could script the creation of such contexts given the bug and its root cause are already known; record/replay is for capturing and reproducing the contexts of previously unknown bugs. The team identified the 21 bugs in Table 3.11.

We empirically compared ATTUNE with KLEE [84], a state-of-the-art test suite generation
tool. We limited our comparison to bugs in Table 3.11 that are part of the coreutils project since KLEE supports coreutils easily. Our other studied bugs have more external libraries, aside from libc, so require additional engineering for KLEE to accommodate. KLEE was given 60 minutes before a timeout ended the test generation, as in [19]. We considered evaluating in comparison to KATCH [117], an extension of KLEE. Still, KATCH has apparently not been maintained since 2014 [163], and the version of KLEE it extended predates the oldest available GitHub release [85].

We show that ATTUNE provides a solution for a wide range of common critical and mundane bugs along with short descriptions about the nature of the specific bug fixes in 3.11.

We also test ATTUNE’s customer-side patch validation leveraging metadata verifying the updated version of the program doesn’t break the customer test suite using a real regression suite while still protecting proprietary source-level information.

3.3.1 ATTUNE successfully validates a wide range of patches

ATTUNE successfully validated the real developer patches for 19 and failed for 2 of the bugs the challenge team collected, marked with ✓ and ✗ in Table 3.11, resp. We organize the 19 bugs successfully handled into several different types and describe how the developer employs ATTUNE in each case, then explain the 2 failures.

**String Parsing** bugs are fairly common as there are often many corner cases, which can have significant security implications since input strings may act as attack vectors. Figure 3.12 [30] adjusts Curl’s treatment of URLs that end in a single colon. In the buggy version, Curl incorrectly throws an error and never initiates a valid http request. The patch modifies one file. Since ATTUNE replaces the entire modified function instead of individual lines of code, it needs to resolve all references in the new version.

ATTUNE uses the recorded execution to recreate the context that triggered the bug, and then jumps to the patched code upon entering the modified function. Since the only change was adding an if statement that doesn’t trigger a recorded event, the ad hoc test continues past the point where the bug occurred, without divergence other than instruction pointer and base pointer. The developer
can set a breakpoint at the patched section, watch the if statement process the input correctly, and verify the string in *portptr. The test then ends since the log has no information regarding how the network would have responded to the HTTP request had it been sent.

Figure 3.13 [32] deals with mishandling URL strings crafted with special characters, e.g., the "#@" in http://example.com#@evil.com caused Curl to send a request to a malicious URL erroneously. The patch calls sscanf with a different filter string. Since the surrounding function handles all the URL parsing for the application, it is rather large with many references. Unlike the above bug, which only requires resolving pointers to old strings, the new filter string must be loaded into a new data section and referenced appropriately. ATTUNE recreates the state that caused the initial behavior and then jumps to the modified code. The developer can verify the patch by checking the values in protobuf and slashbuf.

Mathematical Errors can have security implications when related to pointer errors or integer overflows. For example, a malicious PNG image triggers a bad calculation of row_factor in Figure 3.14 [154], causing a divide-by-zero error and Denial-of-Service (DoS). With traditional bug reports, the user would need to send the image as an attachment, but a legitimate user affected by the DoS is unlikely to be aware of the carefully crafted malicious image uploaded by an attacker - assuming, of course, they are aware of this image uploaded by an attacker. ATTUNE does not require attachments besides the execution trace, since the re-recorded trace includes the image. The developer can use rr’s original replayer to inspect which line caused the error. After the erroneous calculation is localized they write the patch, After the developer writes the patch, they use ATTUNE to verify that row_factor is no longer 0. The patch doesn’t trigger any new events so the function returns gracefully.

We found these types of bugs are fairly common. ATTUNE also successfully replayed patches concerning a calculation error in the Curl library that miscalculated port ranges when a URL specified multiple ports [27] (adds 1 line and removes 1 line in 1 file), and another bug in the libpng library that miscalculated the size of IDAT chunks that make up a png [104] (adds 13 lines and removes 13 lines across 3 files).
New Functions & Function Parameter Refactoring. Many fixes, especially those that pertain to size miscalculations, involve refactoring the buggy function to require a new parameter or writing an entirely new function (with new DWARF and ELF metadata). While not particularly strenuous from the developer’s perspective, these fixes create a challenge from ATTUNE’s perspective. Since both the function that has been refactored or inserted and the functions that call the new/refactored function need to be modified, ATTUNE must replace all these functions in the executable and properly link them.

A patch for the wc file processing utility adds special character parsing functions as shown in Figure 3.15 [181]. ATTUNE loads patched versions of the new function and those functions that call the new function into the address space. The new function is loaded to point towards the original libraries and executables where appropriate, and the modified calling functions point to the new function. There is no need to send a file with the problematic non-standard characters in the bug report to the developer since it is included in the recorded log. These types of bugs can be difficult for conventional bug reports as files in transit may arrive with modified encoding types and changed contents.

ATTUNE provides the input from the recorded file and successfully returns from the modified functions displaying the patched output. Testing the modified wc code doesn’t diverge drastically from the original execution trace. The developer can verify the patch by letting the program run to termination and inspecting the calculated value.

Adding Conditionals. Perhaps the most common patch we saw involved adding conditionals. Many security-critical patches make one-line changes to correct conditional checks. We examined one such example in redis. Such services are particularly hard to test and debug using conventional mocks, as complex network inputs can be difficult to recreate in mocking frameworks. Redis allows monitor connections to send logging and status-checking commands. The buggy version in Figure 3.16 [156] didn’t check the monitor’s client flags, resulting in a kernel panic. While this was one of the smaller patches, the validation process varied substantially from the log. ATTUNE enables the developer to step through the program and watch progress through the modified control
flow past the point of the crash.

**New or Changing Loop Conditions.** Bad loop conditionals are also common. Reference resolution is performed as before, but these patches vary greatly from an ad hoc testing perspective because loop conditionals do not necessarily exhibit the bug on the loop’s first iteration. One such example from the *wget* utility demonstrates how **ATTUNE** handles this sort of change in a security-critical situation. The bug allowed attackers to inject arbitrary HTTP headers via CRLF sequences into the URL’s host subcomponent. Attackers could insert arbitrary cookies and other header info, perhaps granting access to unauthorized resources. The developer modified the *url_parse* functions in Figure 3.17 [183] to check each character in the hostname and throw an appropriate error. During ad hoc testing the developer verifies the patch works by watching the program check each character, and upon entering the if statement, freeing the URL pointer and proceeding correctly to the error handling code.

**Swapped Code:** **ATTUNE** successfully constructed test cases in scenarios that swapped library function calls yes-1 [194] and swapped control flow blocks df-1 [38]. It is important to note that test construction is built from the recorded events, so even though the developer may make far-reaching changes the coverage of the test case is not related to the changes the developer makes. In the case of the yes-1 patch [194] which makes far-reaching changes across the code base to address the same bug in multiple places (15 files). Assuming the recorded log only manifests one instance of the bug, the generated ad hoc test case can only check for that instance, not changes elsewhere in the code base. **ATTUNE** allows for major changes, but the test case constructed actually only covers a small subset of the patch. **ATTUNE** constructs the test case to cover the minimal patch, but if the developer decides to make additional changes they must construct additional test cases by hand.

**Failures:** **ATTUNE** successfully generated ad hoc test cases for those challenge patches where the compiled binaries included complete metadata. However, it failed on **functions with no ELF symbol table entry:** We were initially surprised that a removed break statement in shred-1 [161] caused an error since the change is so small. Upon investigation, we found this behavior should be expected, since the compiler inlined the function (used only in one place) – thus no symbol
table entry for cross-referencing the function. A removed break statement in shred-1 [161] caused a surprising error. While the change is small, the function (used only in one place) is inlined, so there is no symbol table entry. ATTUNE could accommodate in-lining with additional engineering effort. ATTUNE also failed due to **DWARF omissions**: Applying ATTUNE to parameter changes in curl-8 [26] was unsuccessful. We expected to be able to locate the modified function in the loaded binaries to link the patch, but the DWARF metadata generated by the compiler did not include the filename for the file containing that function. ATTUNE depends on the compiler’s compliance with the DWARF specification. ATTUNE also failed due to **DWARF omissions**: Applying ATTUNE to parameter changing in curl-8 [26] was unsuccessful. This appears to be an engineering problem, not an approach limitation. ATTUNE failed to locate pieces of the modified function in the loaded binaries and couldn’t link a patch. This is likely caused by build idiosyncrasies that don’t include the buggy filename in the DWARF information. ATTUNE depends on DWARF information for line numbers so test construction was unsuccessful.

As Table 3.18 shows, ATTUNE created tests for 7 of the 8 studied coreutils bugs, while KLEE created tests for 5 and timed out on 2 that were handled by ATTUNE (wc-2, bs-1). Neither was able to generate tests for the shred-1 bug.

### 3.3.2 ATTUNE’s wait time and memory overhead is small

**Ad Hoc Test Construction Time** ATTUNE’s quilting occurs at load time so runs when each candidate patch is tested. Since ATTUNE’s quilting occurs at load time, the procedure runs when each candidate patch is tested. However, since recording allows for targeted test construction, almost all the overhead introduced by searching the program space is removed. Table 3.18 shows our measurements of quilting overhead relative to KLEE. In all but one case our speedup was well over 90% and could reduce generation time by as much as 99.6%. In absolute time measurements, our worst case was just under 4 seconds. Furthermore, as the program space expands, symbolic execution-based approaches will slow down further while ATTUNE will remain equally fast.

**Memory Footprint:** ATTUNE inserts patched code during testing so it incurs some memory
overhead at test time. However, thanks again to targeted testing this is a one-time operation. Symbolic execution on the other hand requires significant resources to maintain the intermediate program states required to develop test cases. We found on the studied bugs that Attune reduced memory overhead by over 90% in all cases and could reduce memory usage by as much as 97%. Attune inserts patched code during testing so it incurs some memory overhead at test time. However, since Attune patches at the function level, and not the binary level it is extremely lightweight. Table 3.18 shows specific measurements. The worst-case overhead was close to 100KB in curl-2 and a bit over 25KB in curl-11, but otherwise remained well under 25KB. Modern Linux systems with 64-bit address spaces and 4KB page sizes can easily accommodate this overhead.

Increases in the program’s memory footprint from quilting are not related to the size of the patch number of lines of code patched (added or deleted), but instead are related to the size of the functions that have been patched and the associated data those functions access, as well as the number of functions patched. The patch is so large in [31] because it spanned multiple large functions. Memory overhead is tied both to the nature of the patch and the nature of the code base. If the code base contains large monolithic functions, then patching those functions will incur higher overhead, and the more functions the patch spans the higher the overhead will be.

**Patch Sizes.** We want Attune to remain as transparent to the developer as possible so the developer can work unimpeded. While most patches are rather small, our evaluation showed that we could also accommodate rather large code changes (Table 3.11). It is important to note that test construction is built from recorded events, so even though the developer may make far-reaching changes, the test case coverage is not related to the developer’s changes. In the case of [194] which makes far-reaching changes across the code base to address the same bug in multiple places, Attune allows for major changes, but the test case constructed actually only covers a small subset of the patch. Attune constructs the test case to cover the minimal patch, but if the developer decides to make additional changes they must construct additional test cases by hand.
3.3.3 Users validate released patches with their own workloads

**Recording Overhead.** ATTUNE uses a verbose recording system to guarantee bug reproduction, since the replacer relies on the inclusion of register values and results from syscalls, mmapped files, and the specific points at which thread interleaving and signal delivery occurred during the original execution. As such it is not practical for production recording and instead should be used for bug capture in an offline setting as outlined in Figure 5.1. In the worst case ATTUNE shows almost 60x performance and usually presents a 10-20x slowdown. [138] contains complete performance metrics consistent with our findings.

**Execution Trace Sizes.** Our ad hoc test generation workflow assumes that users send execution traces to developers for reproduction and debugging, so we do not want the log files from re-recording to become too large and cumbersome. Figure 3.21 shows exact measurements. We expected a dependency between log sizes and the number of events recorded but found that they are mainly independent. This is because the files used in the execution trace become part of the recording and take up most of the space in the log, not the record of events. The total log size for the studied bugs was always under 5MB.

**User Validation.**

In the last (optional) stage of the patching workflow, the user validates the patch in their own environment to verify no needed functionality has broken. This stage would be particularly useful if combined with our separate static quilting tool described in Chapter 6, enabling users to select bug-fix patches from new releases containing other unrelated changes. Because ATTUNE operates entirely in user space, without hardware, operating system, etc. support, it can run in both developer and user environments. ATTUNE summarizes the “diffs” in source and binary code, and exports metadata along with the released binary patch allowing for user validation.

For sample user environment workloads, we used the Redis benchmark [155], which simulates thousands of different requests to the server, and the httperf benchmarking tool [169] making thousands of connections. The validation procedure for the redis patch [155] is similar to the redis discussion above, but ATTUNE utilizes only the metadata it added to the released patch, shown in
ATTUNE needs to be inserted and deleted line addresses for its runtime decision algorithm. The metadata’s "inserted line addresses" and "deleted line addresses" are offsets into the relevant files while deleted lines from the original binary are balances into the original executable. Inserted lines only appear in the patch release so their addresses are offsets into the patched codefile that gets mapped into memory. As such the overhead of including this information along with the patch is negligible. With this metadata ATTUNE can emulate the user’s original execution trace that demonstrated the bug, to verify it has been fixed, as well as emulate new execution traces from the user’s choice of workloads that do not trigger bugs in the original version. No longer crashes the same way, and then record (with the original) and replay (with the patch) their own test suite workloads. ATTUNE verified the patch by emulating the test (generated from the buggy recording) the same way it did in the developer environment, and verified no additional functionality was broken when the recorded workload executed with the same results it had during recording.

3.3.4 Threats to Validity

Internal. As far as we know, no execution traces were recorded when any of the studied bugs were discovered. Some of our scripts for recording the buggy version run bug reproduction tests included in the real bug reports, but others were contrived. This threat is partially mitigated since the contrived scenarios were developed by three grad students who were not ATTUNE developers. We describe how we imagine a developer would verify patches using ATTUNE, but we are not developers on these projects and lack the developers’ knowledge. This is mitigated to some extent since ATTUNE generated ad hoc tests for the real developer patches. Lastly, since do not have execution traces for any real users using the programs in our dataset, we simulated workloads with benchmarks that may not be representative of how real users would validate these programs. Running the real regression test suite and using muplay is explained in the introduction, we don’t need to repeat it here. The actual developer may look to other places in the code, or require a more thorough examination to deem a patch as valid. We also don’t know other considerations.
the real developers consider before a patch is published. This could include performance metrics, 
continuous integration, and static analysis checks. In our evaluation of ATTUNE we didn’t try to 
simulate these common tactics for many professional-level projects.

**External.** We demonstrate that ATTUNE supports various single-line and multi-line patches 
for security vulnerabilities and other bugs in real programs. ATTUNE resolved references between 
modified and original executables and program state with binary transformations, but we cannot 
claim that ATTUNE’s set of transformations will resolve all types of references supported by the 
expansive x86-64 instruction set. We have not yet studied C++ or other non-C programs or 
investigated ARM or other architectures. The bugs we studied may not represent real-world bugs; 
notably, we have not yet studied GUI bugs. In principle, rr supports programs written in any 
language, but so far we have only studied C programs. - what rr can do is irrelevant

3.3.5 Limitations

Our ATTUNE prototype extending rr inherits rr’s design decision to replay multi-threaded 
recordings on a single thread and simulate thread interleaving by interrupting that single thread’s 
execution [138, 137]. Although ATTUNE accommodates thread synchronization and faithfully 
emulates the error state, rr’s approach makes it impossible for ATTUNE to accurately verify patches 
for concurrency bugs that manifest due to the true parallelism of multi-core execution. There is 
nothing in ATTUNE itself that inherently prevents it from addressing concurrency bugs, but we 
would need to find a faithfully multi-threading replacement for rr. Our ATTUNE prototype inherits 
rr’s replay paradigm which runs the recording on a single thread during replay, so replayed parallel 
programs run only on a single core [138, 137]. ATTUNE accommodates thread synchronization, and 
faithfully emulates the error state. Still, because rr simulates thread interleavings by interrupting a 
single thread execution, ATTUNE cannot accurately verify patches for concurrency bugs.

ATTUNE also relies on rr to re-record the execution trace in the user environment and to replay 
that recording in the developer environment with the original version of the program [138, 137, 
129]. Since rr was designed to be used during developer testing, with too high overhead for
production [138], we adopt the re-recording model shown in Figure 3.2. In theory, lightweight production recorders could fail to capture sufficient detail to faithfully replay some behaviors even in the same user environment, in which case the re-recording might not manifest the bug. Still, Mashtizadeh et al. [118] explain this limitation is generally unimportant in practice.

A few ATTUNE limitations are orthogonal to the recorder. ATTUNE does not currently verify patches to preprocessor macros, since it compares the source file versions rather than the results of preprocessing the source files. ATTUNE also does not currently support generating tests for patches that change the size of a data structure on the stack or in the heap. We allow new values to be put on the stack and heap but don’t adjust memory allocation when replaying logged values.

Independent of rr limitations, ATTUNE does not support data structure changes that change the size of a struct on the stack or in the heap, which would require changes to memory allocation. Anything that modifies values stored on the heap or the stack during execution is not emulated. This mainly pertains to fixes that modify the contents of a struct or a class. We allow new values to be put on the stack and heap, but we don’t change the memory allocation when replaying from the recorded log. ATTUNE also does not verify patches to preprocessor macros, which are inserted inline when executables are generated; there are no associated symbols so a macro cannot be replaced in the same way that ATTUNE replaces functions. In principle, ATTUNE could be augmented to instead replace each function that calls a changed macro, an engineering effort.
<table>
<thead>
<tr>
<th>Bug</th>
<th>Success or Failure</th>
<th>Patching Effort</th>
<th>Files Modified</th>
<th>LOC Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl-1 [30]</td>
<td>✓</td>
<td>Changes how a string is parsed</td>
<td>1</td>
<td>16+, 16-</td>
</tr>
<tr>
<td>curl-2 [31]</td>
<td>✓</td>
<td>Changes function arguments and calls</td>
<td>4</td>
<td>9+, 9-</td>
</tr>
<tr>
<td>curl-5 [28]</td>
<td>✓</td>
<td>Modified if statement for buffer overflow</td>
<td>1</td>
<td>4+, 1-</td>
</tr>
<tr>
<td>curl-6 [29]</td>
<td>✓</td>
<td>Added new function and inserted call</td>
<td>1</td>
<td>55+, 8-</td>
</tr>
<tr>
<td>curl-8 [26]</td>
<td>✗</td>
<td>Changes multiple functions calling a write function</td>
<td>4</td>
<td>17+, 17-</td>
</tr>
<tr>
<td>curl-9 [33]</td>
<td>✓</td>
<td>Change parameters to a function call</td>
<td>1</td>
<td>2+, 1-</td>
</tr>
<tr>
<td>curl-10 [173]</td>
<td>✓</td>
<td>Adds a condition check</td>
<td>1</td>
<td>6+, 2-</td>
</tr>
<tr>
<td>curl-11 [27]</td>
<td>✓</td>
<td>Off by 1 correction</td>
<td>1</td>
<td>1+, 1-</td>
</tr>
<tr>
<td>curl-12 [32]</td>
<td>✓</td>
<td>Changes libc calls to add extra parsing</td>
<td>1</td>
<td>5+, 5-</td>
</tr>
<tr>
<td>libpng-1 [154]</td>
<td>✓</td>
<td>Calculation modification for divide by 0 error</td>
<td>1</td>
<td>6+, 3-</td>
</tr>
<tr>
<td>libpng-2 [104]</td>
<td>✓</td>
<td>Adjust calculation for idat chunk max</td>
<td>3</td>
<td>13+, 13-</td>
</tr>
<tr>
<td>wc-1 [181]</td>
<td>✓</td>
<td>Added new function and changed condition check</td>
<td>1</td>
<td>23+, 2-</td>
</tr>
<tr>
<td>wc-2 [180]</td>
<td>✓</td>
<td>Added error condition check</td>
<td>1</td>
<td>3+, 0-</td>
</tr>
<tr>
<td>yes-1 [194]</td>
<td>✓</td>
<td>Substantial changes in option parsing</td>
<td>15</td>
<td>40+, 141-</td>
</tr>
<tr>
<td>shred-1 [161]</td>
<td>✗</td>
<td>Removed a break statement</td>
<td>1</td>
<td>1+, 1-</td>
</tr>
<tr>
<td>ls-1 [111]</td>
<td>✓</td>
<td>Added condition for change in option parsing</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>mv-1 [1]</td>
<td>✓</td>
<td>Adding a conditional check before operation</td>
<td>2</td>
<td>6+, 0-</td>
</tr>
<tr>
<td>df-1 [38]</td>
<td>✓</td>
<td>Replacing open calls with stat calls</td>
<td>2</td>
<td>12+, 8-</td>
</tr>
<tr>
<td>bs-1 [157]</td>
<td>✓</td>
<td>Changing a loop condition</td>
<td>1</td>
<td>2+, 1-</td>
</tr>
<tr>
<td>wget-1 [162]</td>
<td>✓</td>
<td>Adding conditional check for log</td>
<td>1</td>
<td>1-, 2+</td>
</tr>
<tr>
<td>wget-2 [183]</td>
<td>✓</td>
<td>Adding for loop to free variables no longer needed</td>
<td>1</td>
<td>11+, 0-</td>
</tr>
<tr>
<td>redis-1 [156]</td>
<td>✓</td>
<td>Adding conditional check</td>
<td>1</td>
<td>1+, 1-</td>
</tr>
</tbody>
</table>

**Figure 3.11:** Patch-Testing Dataset
+ if(!portptr[1]) {
+   *portptr = '\0';
+   return CURLUE_OK;
+ }
- if(rest != &portptr[1]) { ...
- ...
+ *portptr++ = '\0'; /* cut off the name there */
+ *rest = 0;
+ msnprintf(portbuf, sizeof(portbuf), "%ld", port);
+ u->portnum = port;
...

Figure 3.12: Curl-1 URL Parsing

static CURLcode parseurlandfillconn(...) {
    path[0]=0;
    rc = sscanf(data->change.url,
-   "%15[^n:]:%3[/^n/?:]%[^n]",
-   "%15[^n:]:%3[/^n/?:]%[^n]", /*new data*/
+   protobuf, slashbuf, conn->host.name, path);
    if(2 == rc) {
    ...

Figure 3.13: Curl-12 String Parsing

png_check_chunk_length (...) {
...
    size_t row_factor =
-      (png_ptr->width * png_ptr->channels
-       * (png_ptr->bit_depth > 8? 2: 1)
-       + 1 + (png_ptr->interlaced? 6: 0)));
+      (size_t)png_ptr->width
+      * (size_t)png_ptr->channels
+      * (png_ptr->bit_depth > 8? 2: 1)
+      + 1
+      + (png_ptr->interlaced? 6: 0);

Figure 3.14: libpng-1 Mathematical Error
Return non zero if a non breaking space. */
+ static int iswnbspace (wint_t wc) {
+ return ! posixly_correct && (wc == 0x00A0 ...  
+ static int isnbspace (int c) {
+ return iswnbspace (btowc (c));
+
+ wc (args) {
- if (iswspace (wide_char))
+ if (iswspace (wide_char)
  || iswnbspace(wide_char))
    goto mb_word_separator;
...
- if (isspace (to_uchar (p[-1])))
+ if (isspace (to_uchar (p[-1]))
  || isnbspace (to_uchar (p[-1])))
  || isnbspace (to_uchar (p[-1])))
+ goto word_separator;
}
...

Figure 3.15: wc-1 New Function and Refactoring

void addReplyErrorLength
  (client *c, const char *s ... )
{
- if (c->flags & (CLIENT_MASTER|CLIENT_SLAVE)) {
+ if (c->flags & (CLIENT_MASTER|CLIENT_SLAVE)
  && !(c->flags & CLIENT_MONITOR)) {
    char* to = c->flags &
    CLIENT_MASTER? "master": "replica";
...  

Figure 3.16: redis-1 Erroneous Conditional

url_parse (const char *url ...) {
...
+ /* check for invalid control characters in host
   name */
+ for (p = u->host; *p; p++) {
+ if (c_iscntrl(*p)) {
+ url_free(u);
+ error_code = PE_INVALID_HOST_NAME;
+ goto error;
+ }  
+ }

Figure 3.17: wget-2 New Loop
<table>
<thead>
<tr>
<th>Bug</th>
<th>ATTUNE</th>
<th>KLEE</th>
<th>ATTUNE Time</th>
<th>KLEE Time</th>
<th>Speedup</th>
<th>ATTUNE Mem</th>
<th>KLEE Mem</th>
<th>Overhead Cut</th>
</tr>
</thead>
<tbody>
<tr>
<td>wc-1 [181]</td>
<td>✓</td>
<td>✓</td>
<td>1.37s</td>
<td>300.046s (5m)</td>
<td>99.5%</td>
<td>5.9 KB</td>
<td>108.388 KB</td>
<td>94.5%</td>
</tr>
<tr>
<td>wc-2 [180]</td>
<td>✓</td>
<td>✗</td>
<td>1.277s</td>
<td>na</td>
<td>na</td>
<td>2.8 KB</td>
<td>107.7 KB</td>
<td>97.4%</td>
</tr>
<tr>
<td>yes-1 [194]</td>
<td>✓</td>
<td>✓</td>
<td>3.4s</td>
<td>8.569s</td>
<td>60.3%</td>
<td>10.6 KB</td>
<td>107.09 KB</td>
<td>90.1%</td>
</tr>
<tr>
<td>shred-1 [161]</td>
<td>✗</td>
<td>✗</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>ls-1 [111]</td>
<td>✓</td>
<td>✓</td>
<td>1.6s</td>
<td>19.57s</td>
<td>91.82%</td>
<td>7.4 KB</td>
<td>132.9 KB</td>
<td>94.4%</td>
</tr>
<tr>
<td>mv-1 [11]</td>
<td>✓</td>
<td>✓</td>
<td>3.6s</td>
<td>58.4s</td>
<td>93.84%</td>
<td>4.3 KB</td>
<td>208.2 KB</td>
<td>97%</td>
</tr>
<tr>
<td>df-1 [38]</td>
<td>✓</td>
<td>✓</td>
<td>1.48s</td>
<td>18.869s</td>
<td>92.15%</td>
<td>5.97 KB</td>
<td>151 KB</td>
<td>96.05%</td>
</tr>
<tr>
<td>bs-1 [157]</td>
<td>✓</td>
<td>✗</td>
<td>1.2s</td>
<td>na</td>
<td>na</td>
<td>5.6 KB</td>
<td>113.37 KB</td>
<td>95.06%</td>
</tr>
</tbody>
</table>

**Figure 3.18:** Comparison to KLEE Test Generation

![Comparison to KLEE Test Generation](image)

**Figure 3.19:** Quilting Memory Footprint

![Quilting Memory Footprint](image)

**Figure 3.20:** Patch Sizes

![Patch Sizes](image)
Figure 3.21: Execution Trace Sizes

[fontsize=\footnotesize]
inserted line addresses:
  0x6b
  0x6e
deleted line addresses:
  0x495AD
  0x495B7
patched code:
...  
  69: jne 0xb9
  6b: and 0x2,%eax
  6e: lea -0x58090939(%rip),%rdx
  75: mov 0x58(%rbx),%rax
...  

Figure 3.22: redis-bug-1 Metadata for User Validation
3.4 Conclusion

**ATTUNE** (**Ad hoc Test generation ThroUgh biNary rEwriting**) leverages record-replay and binary rewriting technologies to automate test generation for security vulnerabilities and other critical bugs discovered post-deployment, when there are no existing tests for testing candidate patches, and little time for constructing and vetting new tests. **ATTUNE** first quilts the modified functions (the patch) into the original binary and then interprets the recorded execution trace from the original binary, as it is executed in the user environment, to “replay” on the patched binary in the developer environment. The developer monitors the progress of the ad hoc test to check that the bug no longer manifests, but does not intervene in test generation and does not need to build test scaffolding. **ATTUNE** also produces metadata that the developer can deploy with the patched program, which enables users to validate the patch by using **ATTUNE** to create additional ad hoc tests from their own workloads. We showed that **ATTUNE** successfully generates tests for a wide range of known security vulnerabilities and other bugs in recent versions of open-source software, with minimal developer effort, both quickly and efficiently. We also showed that **ATTUNE** does so with minimal interference during the debugging and patching process. Lastly, we showed that while **ATTUNE**’s verbose recording mechanism incurs significant overhead, it requires minimal client interference and maintenance during recording and simulated client validation. Our open-source implementation is available at https://github.com/Programming-Systems-Lab/ATTUNE.
Chapter 4: Decompiled Identifier Renaming Engine using Contextual Transformers
4.1 Introduction

Software reverse engineering is the process of analyzing software binaries to extract information about their design and implementation. It finds applications in fields like legacy software maintenance, malware analysis, and cyber forensics. During code inspection, a security or software analyst typically has access to only a binary executable (no high-level source code), especially for proprietary software. Without human-comprehensible information like meaningful variable names, it becomes difficult for the analyst to understand and reason about the binaries, as all high-level code abstractions are lost. So analysts employ tools that convert binary executables into a form that can be more easily comprehended.

In instances like legacy software maintenance, malware analysis, and cyber forensics analysts must comprehend code that they didn’t write, can’t build, and don’t have the source for. This process is known as software reverse engineering.

For example, reverse engineering of the Solar Winds attack in 2020 [55] identified the root cause as a tainted version of the Orion software resulting in immediate updates. Similarly, reverse engineering uncovered the rebirth of ZeUS malware variants during the coronavirus pandemic of 2020 so antivirus and intrusion detection systems could be updated.

In consumer protection, reverse engineering ByteDance’s TikTok app revealed intrusive data mining resulting in the US government placing unprecedented sanctions on the Chinese company. Reverse engineering has even proved useful in feature design as Dropbox famously reverse-engineered portions of Apple’s operating system so their filesharing platform would integrate with the OSX file explorer. As a result, Apple attempted to acquire the smaller startup.

Proprietary software often comes in binary form, making it difficult to comprehend its functionality, as many high-level code abstractions (e.g., meaningful variable names, code structures, etc.) are lost when source code is compiled to binaries. To extract meaningful information from binaries, software analysts typically use reverse engineering that converts binary executables into another form that can be more easily comprehended [42]. Reverse engineering is often applied
void __fastcall add_match(char *a1) {
    // ... var declarations omitted ...
    v1 = (int)(a1 - 1);
    while ( 1 ) {
        v3 = *(unsigned __int8 *)(v1++ + 1);
        v2 = v3;
        if ( !v3 ) break;
        v4 = v2 > 0x7F;
        if ( v2 != 127 )
            v4 = v2 > 0x1F;
        if ( !v4 ) {
            free(a1);
            return;
        }
    }
    // ... some code omitted ...
}

Figure 4.1: Real world Hex-Rays decompilation. The reconstructed source differs significantly from the original, and it is hard to deduce the original developers' intentions.

in binary code inspection, legacy software maintenance, malware analysis, and cyber forensics. For example, reverse engineering uncovered the rebirth of ZeUS malware variants during the coronavirus pandemic of 2020 [142].

Traditionally, the primary reverse engineering tools are disassemblers, which extract assembly instructions from a binary executable. However, in recent years, decompilers like Ghidra [53] and Hex-Rays [64] have become practical and popular. They produce a source code-like approximation of the binary code as shown in Figure 4.1. While these tools can retrieve the approximate code structure, they introduce variable names that have no semantic meaning, drastically reducing code readability and comprehensibility [80, 67, 61].

However, since compilation wipes away the developer-assigned variable names, retrieving this information from binary is challenging. Traditionally, decompilers are used for this purpose [53, 64]. For example, a malware analyst often decompiles a binary code to comprehend it. However, variable names in the decompiled code often contain dummy variables that lend no semantic meaning to their functionalities, leaving decompiled code difficult to work with and hard to interpret.
Recently, Machine Learning (ML) based techniques [97] have been applied to good effect to this problem as well as related software engineering tasks including code completion, malware analysis, automated program repair, and documentation generation [3].

Machine learning models have proved useful because of their ability to discern patterns from relatively unstructured data. Since code can be regarded as a sequence of tokens chosen from some vocabulary, models used for processing natural language can be adapted to process code. However, code differs from natural language in some significant ways, and hence these models require some modifications to be applied most effectively [40]. Building a machine learning model specifically for decompiled code is challenging because variable names are the primary source of semantic information about code, and decompiled code does not contain this information. Still, some Machine Learning-based models have shown promise in recovering lost variable names from decompiled code using a frequency-based model [62] or LSTMs [97]. However, variables in source code are not independent of each other and often have hidden long-range dependencies. While the NLP community has recently shifted away from LSTMs in favor of context-rich transformer-based representations for other reasons, it turns out LSTMs and frequency-based models are not well-suited to capture such dependencies. Since transformer-based models can more effectively capture long-range dependencies [176], in this work we explore transformer-based models to recover variable names from decompiled code.

As mentioned above transformers have become popular in natural language processing [176, 36, 192]. However, code differs from natural language in many significant ways [3, 40], hence vanilla transformer architectures need modifications for practical application to the task of variable recovery. Consider the following domain-specific aspects of the problem:

**Unknown number of tokens to be predicted:** Transformers that capture bidirectional context usually predict a known number of tokens, but to make the vocabulary size manageable, identifiers must be split into subtokens. For example, an identifier like `my_var` could be split up as three subtokens - “my”, “_”, and “var”. Each identifier can be comprised of an arbitrary number of subtokens, and the model needs to access the information contained in the entire sequence
while predicting the name of an identifier. To deal with this problem, we use an encoder-decoder transformer architecture as in [2].

**Syntactic constraints:** Unlike natural language, code’s strict syntax requires that a variable assigned a name at one occurrence in the prediction must be the same at all other occurrences. For example, if a decompiled identifier name \( v1 \) appears on line 3 and line 100, the predicted name must be the same on both lines. We propose a novel algorithm that uses the joint probability over sequences to predict variable name identifiers while still obeying constraints imposed by the code syntax.

**Token Non-uniformity:** While training a model for natural language, all tokens are usually given equal importance [176]. However, for the semantic understanding of code, the identifier tokens are more important than those tokens that are built into the language syntax. For example, a variable name like “\texttt{click\_count}” provides much more semantic information than a keyword like ”\texttt{while}”. We propose a token weighting scheme specially crafted for the variable name recovery problem.

**Code sequences are long:** Adaptations of NLP techniques to code often consider functions analogous to sentences. Traditional transformers limit the maximum sequence size to a few hundred tokens. While this restriction rarely presents a problem dealing with sentences, many functions are much longer. For example, the longest function in our benchmark datasets is over 4000 tokens long. To handle longer functions we propose a mechanism to break long sequences into smaller pieces and recombine their individual predictions while still obeying code’s syntactic constraints.

Putting all these together, we propose **DIRECT** (Decompiled Identifier Renaming Engine using Contextual Transformers), the first transformer-based model built specially for variable recovery from decompiled binaries.

We compare **DIRECT** to **DIRE** [97], the state of the art in variable name recovery on a benchmark dataset, and show that **DIRECT** improves on the baseline by 20%. We also evaluate the individual impact of each of our specific adaptations by performing a series of ablation studies. We provide
the source code for DIRECT \(^1\) in the hope that it will prove to be a useful tool for other researchers.

In this work we conduct an empirical case study to illustrate that transformer-based ML represents a step forward in the state-of-the-art modeling of decompiled binaries, but only after a series of custom modifications designed specifically to handle decompiled code as opposed to natural language. We also present empirical evidence that shows the effectiveness of our techniques on the benchmark dataset as well as ablation studies demonstrating the effectiveness of each modification. DIRECT achieves a 20\% improvement in accuracy over the previous state of the art and also demonstrates gains on three other common metrics. We hope that in addition to providing an open-sourced practical tool, this work helps present solutions to some of the challenges researchers face while adapting transformers to solve other code-related problems.

We hope that in addition to providing an open-sourced practical tool for decompiled identifier renaming, our work will provide useful guidelines for future applications of transformer models to code.

\(^1\)https://github.com/DIRECT-team/DIRECT-nlp4prog
4.2 Design

Figure 4.2 provides an overview of DIRECT. In this section we detail each of the problems we encountered and the design decision solutions.

4.2.1 Encoding/Decoding

Transformers are traditionally used to predict entire sequences; however, in our problem setting most tokens are fixed. Therefore we need to adapt transformers from predicting entire sequences to predicting individual tokens based on the fixed tokens.

While making a prediction on the occurrence of a particular variable, the model should ideally have access to the information contained in the entire input sequence. The naive solution is to use a bidirectional transformer with a Masked Language Model (MLM) training scheme, such as BERT. However, by design, an MLM is designed to predict the same number of tokens as in the input sequence. In our case, because of sub-tokenization, the predicted subsequences can be of unknown length.
length. Adapting an MLM transformer to solve this problem is non-trivial.

The next option is to use a transformer as a sequence-to-sequence language model to predict the immediate next token given all the preceding tokens. One could feed the entire sequence until a variable is reached, start generating tokens one at a time in an autoregressive manner, and stop when a special end token is predicted. However, such a model cannot use bidirectional context while making a prediction, and can only leverage the part of the sequence that precedes each variable.

This problem of bidirectional context plus variable length blanks is relatively under-explored in the NLP community. Blank Language Models [160], a recently proposed approach, uses a bidirectional transformer to generate an embedding of a blank and then uses this embedding to predict both a word and an optional blank alongside it. This is then fed back into the transformer to produce new embeddings and this repeats until all the blanks are filled.

This model cannot be directly applied to our scenario either, because blanks corresponding to the same variable must have the same prediction. While one could conceivably find a workaround, we choose instead a different approach that is specifically tailored to our problem.

We propose to use an encoder-decoder setup, as in [176]. The transformer encoder embeds each input token, and the sequential decoder attends over these encoder embeddings while making predictions one token at a time. So although we give the decoder only the portion of the input sequence that precedes the variable of interest, it also has access to the entire input sequence through the encoder embeddings.

Of course, this still leaves open the question of how to constrain multiple instances of the same variable to have the same prediction. While we will present a better solution to this problem in Section 4.2.2, a good first approximation is to simply use the prediction at the first occurrence of the variable we are interested in. We hypothesize that since the encoder-decoder model has access to the entire sentence while making a prediction for each occurrence, one cannot do drastically better than this simple approximation.
4.2.2 Advanced Prediction Algorithm

Effective sequence modeling requires not only making predictions but also predictions that fit the problem setting [40]. Semantic preserving identifier renaming mandates that once a variable has been renamed it must have the same value at each occurrence. This additional constraint poses a challenge for vanilla transformers since they predict each token independently in traditional language modeling. Exhaustively searching the target vocabulary space is computationally intractable, so we narrow the search space with a specialized prediction algorithm that fits the problem set.

At the variable’s first occurrence, we make \( m \) predictions for its name, each of which leads to a different sequence of variable name assignments. Throughout our algorithm, we maintain the top \( k \) sequences only. Thus at the first occurrence of a variable, we generate \( m \times k \) possible sequences and pick the top \( k \). In practice, we use \( m = k \).

At later occurrences of a variable, we update the scores of the existing predictions, thus maintaining the list of \( k \) sequences. This is where our algorithm differs from standard beam search. Note that the predictions made at the first occurrence of a variable constrain its predictions at further occurrences, but choosing a large \( k \) mitigates this problem.

This procedure, “Advanced prediction”, is shown in Figure 4.3 for the case when \( k = 2 \). Algorithm 3 describes it in detail. In our experiments, we observed that choosing \( k = 5 \) was optimal.
Algorithm 3: Advanced Prediction

Input: A sequence of decompiler output tokens $S$, and a model $M$

Output: $S$ with predicted names

$\text{gen} \leftarrow \text{[]}$, $\text{probs} \leftarrow [1]$;

$\text{foreach } tok \in S \text{ do}$

$\quad \text{if } tok \text{ is not a variable then}$

$\quad \quad \text{foreach } seq \in \text{gen} \text{ do}$

$\quad \quad \quad \text{seq.append}(tok)$;

$\quad \quad \text{end}$

$\quad \quad \text{continue};$

$\quad \text{end}$

$\quad \text{if } tok \text{ has been seen before then}$

$\quad \quad \text{foreach } j \in 1...\text{len}(\text{gen}) \text{ do}$

$\quad \quad \quad n \leftarrow \text{current pred of } tok \text{ in } \text{gen}[j]$;

$\quad \quad \quad p \leftarrow \text{prob assigned to } n \text{ by } M \text{ at the current position}$;

$\quad \quad \quad \text{gen}[j] \leftarrow \text{gen}[j] + n$;

$\quad \quad \quad \text{probs}[j] \leftarrow \text{probs}[j] \times p$;

$\quad \text{end}$

$\text{end}$

$\text{foreach } j \in 1...\text{len}(\text{gen}) \text{ do}$

$\quad \text{Beam search sub-tokens with } M, \text{ find the top } k \text{ possibilities for the name of } tok;$

$\quad \text{Let the names be } n_1, ..., n_k \text{ and their probabilities be } p_1, ..., p_k;$

$\quad \text{Replace } \text{gen}[j] \text{ with } (\text{gen}[j] + n_1), ..., (\text{gen}[j] + n_k);$;

$\quad \text{Replace } \text{probs}[j] \text{ with } (\text{probs}[j] \cdot p_1), ..., (\text{probs}[j] \cdot p_k);$;

$\text{end}$

Sort $\text{gen}$ and $\text{probs}$ in descending order of $\text{probs}$;

$\quad \text{gen} \leftarrow \text{gen}[1:k]$;

$\quad \text{probs} \leftarrow \text{probs}[1:k]$;

$\text{end}$

return $\text{gen}[1]$;
Figure 4.3: Advanced Prediction with $k = 2$. The decoder takes as input the portion of the sequence that precedes the variable being predicted. Our algorithm differs from standard beam search in the prediction of the second occurrence of $v1$. Rather than generate multiple predictions for $v1$, the algorithm simply updates the scores of the existing predictions in order to obey the syntactic constraints of code.
4.2.3 Identifier Token Coefficient

A typical transformer treats all tokens identically when computing the loss function during pre-training and fine-tuning. Code differs from natural language in that grammar requires the majority of the tokens. The only opportunity for the programmer to inject semantic meaning into the source code text is through identifiers, which makes this problem compelling in the first place. The model should therefore treat identifier tokens differently.

We implement this concept by training with a custom loss function as shown in Figure 4.4. Traditional NLP architectures predict the entire sequence and then train on a loss function by averaging the error uniformly across all tokens. Our custom weighting scheme places increased significance on prediction of identifiers, using a mask which increases the loss 50-fold for identifiers as compared to all other tokens. We expect that this identifier token coefficient (ITC) hyperparameter could be tuned in the future for better performance.

Predicting the identifiers and ignoring the rest of the characters in the sequence would result in a model that doesn’t learn the context surrounding the identifier which informs the prediction.

4.2.4 Splitting and Merging Mechanism

Another inherent difference between code and natural language when considering sequence-to-sequence modeling is the length of the sequence. Discussion of natural language modeling overlooks this aspect since sentences rarely exceed 200 tokens. In code, however, functions are significantly longer so the ML models must support sequences of arbitrary length. In fact, our benchmark dataset contains a small number of sequences with lengths greater than 2000. With respect to identifier recovery, longer sequences mean more variables to recover, multiple usages per variable, and more opportunity for errors. This poses a problem for transformers as traditional transformer-based architectures, like BERT, require a maximum sequence length set in advance. Furthermore, since attention must be trained across all tokens, memory usage increases quadratically with sequence length.

In order to use our model for arbitrary sequence lengths, we developed a novel splitting and joint
Figure 4.4: Identifier Token Coefficient loss function. The Negative Log Likelihood (NLL) loss is computed for each token, and a weighted sum is taken to compute the loss.
prediction mechanism. As described in Figure 4.2 we divide the sequence into multiple chunks of 512 tokens upon which the model predicts. A single variable can have a different prediction in each chunk we combine these predictions using the prediction at the first chunk in which a variable occurs.

We also tried using the chunk with the highest confidence, but we found that this did not perform as well. We suspect this is because the probabilities are less than one, and multiplications with each successive variable only decrease the probability of the entire sequence. Hence smaller pieces with perhaps just one or two occurrences of a variable will be more confident in their predictions despite having less information. One could impose a penalty for pieces with fewer variables, but we defer this analysis to future work.

Other transformer variants can handle sequences of arbitrary lengths like XLNet [192], and we expect these advanced models will handle this issue as well as present new challenges. We again leave these endeavors to future work.

4.2.5 Totality

Using the techniques from the previous sections, we put it all together to get DIRECT, a state-of-the-art variable renaming system. Given an input sequence, DIRECT splits it into pieces of length at most 512 each (default BERT architecture), and puts each piece through a BERT encoder and a BERT decoder with advanced prediction (Algorithm 3). Different predictions across pieces for the same variable are combined by taking the prediction of the first piece in the sequence that contains the variable. Figure 4.2 depicts the entire model.
4.3 Experimental Setup

4.3.1 Data

We use the dataset provided by DIRE [97]. It was generated using C binaries from Github, which were then decompiled using Idas Hex-Rays decompilation plugin [64]. The training data set consists of 1,011,049 functions, with a median of 16 variables per function, a median of 4 unique variables per function, and a median sequence length of 150 subtokens. We follow DIRE and use Sentencepiece [92] to split the functions into subtokens.

We use only the “Body-not-in-train” subset for the validation and test data. They consist of 23662 and 24862 examples, respectively.

4.3.2 Metrics

We define accuracy as an exact match between the original variable name as determined by the debug information mapping, and the name predicted by DIRE. We also examine the edit distance between predicted names and true names and use the edit distance per number of characters (the character error rate) as our metric as in DIRE [97] to capture the success of partial matches. We also measure the Jaccard similarity which is the ratio of the number of overlapping n-grams between two sequences to the total number of n-grams contained in them. We use n=1, so that each word is treated as a set of its constituent characters. There are some instances when decompiler variables have no corresponding true name. These are ignored from all metrics.

4.3.3 Pre-training Procedure

We pre-train one BERT model using the standard MLM task on source sequences directly from the decompiler output (with the dummy variable names from the decompiler). We call this the BERT encoder. Similarly, we pre-train another BERT model using MLM on target sentences (with the true variable names) and call this the BERT decoder. Both models used 4 attention heads, 6 hidden layers, and a hidden embedding size of 256. We trained the encoder and decoder for
220k and 140k steps, respectively, using a batch size of 128 sequences. While masking tokens, we do not differentiate between variable and non-variable tokens since we want the model to learn the complete structure of the code sequences. We also used the standard optimization techniques employed by BERT [36], wherein an Adam optimizer is used with a variable learning rate. The learning rate increases linearly from 0 to $10^{-4}$ over the "warm-up" period of 40k iterations and then decreases linearly from $10^{-4}$ to 0 at the end of pre-training.

4.3.4 Fine-tuning Procedure

After reviewing our proof of concept experiments we trained our best configuration for 85 epochs to produce the DIRECT prototype. We follow the same convention as DIRE [97], whereby the number of sequences per batch is variable, but the total number of tokens in the batch is fixed to define the size of the batch. We used a batch size of 4096 tokens per batch. We used a learning rate of 1e-4 for the first 10 epochs, 0.3e-4 for the next 10, and 1e-5 thereafter.
<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy (%) ↑</th>
<th>Top-5 Accuracy (%) ↑</th>
<th>CER ↓</th>
<th>Jaccard Dist ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRE</td>
<td>35.8</td>
<td>41.5</td>
<td>.664</td>
<td>.537</td>
</tr>
<tr>
<td>DIRECT</td>
<td>42.8</td>
<td>49.3</td>
<td>.663</td>
<td>.501</td>
</tr>
<tr>
<td>Improvement</td>
<td>20%</td>
<td>19%</td>
<td>.2%</td>
<td>6.5%</td>
</tr>
</tbody>
</table>

Table 4.1: Test Accuracy, Top-5 Accuracy (computed by taking the top 5 predictions for each sequence and using the predictions of variables contained in these sequences), Character Error Rate and Jaccard distance of DIRE vs DIRECT. DIRECT outperforms DIRE on all four metrics. DIRE results are reproduced by re-running the authors’ code on our dataset.

### 4.4 Results

#### 4.4.1 DIRECT Evaluation

In order to evaluate the effectiveness of DIRECT, we compare its performance against that of DIRE on our test dataset. The results are shown in Table 4.1. We observe that DIRECT achieves an increase of 7.1 percentage points in accuracy over DIRE, which is a relative increase of 19.9%. We obtained all DIRE results by re-running the authors’ code on the dataset, rather than simply using the numbers from their paper.

#### 4.4.2 Qualitative Analysis

We also perform some qualitative inspection of the attention weights of the trained model to understand what information it is using to make its inferences. An example of this is shown in Figure 4.5 where the predicted identifier is outlined in black. The attentions shown are the weights used while predicting a name for the variable shown in a box, averaged over all attention heads at the last layer of the decoder.

We observe that when making a prediction on the first occurrence of a variable, the decoder model pays attention mainly to the function header, more specifically the return type and function name. However for later occurrences of the same variable, although it does look at the function header to some extent, it relies chiefly on its predictions for earlier instances of the same variable.
Figure 4.5: A visualization of the attention weights of the trained decoder while predicting variables. Darker represents larger weights. The variable subtokens that are being predicted are boxed. For more details, refer to Section 4.2.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy (%) ↑</th>
<th>CER ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform token weighting</td>
<td>30.0</td>
<td>.80</td>
</tr>
<tr>
<td>Weighting identifiers only</td>
<td>33.7</td>
<td>.76</td>
</tr>
<tr>
<td>ITC weighting scheme</td>
<td>34.4</td>
<td>.75</td>
</tr>
</tbody>
</table>

Table 4.2: Validation accuracy and Character Error Rate for various token weighting schemes. Prediction is done using the “first prediction” strategy. All the models are trained for 15 epochs. Refer to Section 4.2 for more details.

4.4.3 Performance on Long Sequences

The graph in Figure 4.6 shows the accuracy of DIRECT on sequences of various lengths. As we cross the 500 token mark, and the splitting technique takes over, there is a steep drop in accuracy. This problem is mirrored in DIRE’s accuracy although not quite as steeply. Still for sequences of length less than 512 tokens DIRECT has an improvement of 10 percentage points over DIRE (48.9% vs. 38.8%). DIRE has high accuracy in the longest two sets of sequences, but this is likely an anomaly caused by insufficient sample sizes.

Other transformer-based variants address this sequencing issue such as XLNet [192], and we expect these advanced models will handle this issue as well as present new challenges. We again leave these endeavors to future work.

4.4.4 Ablation Studies

In this section, we evaluate the impact of each of our design choices. We train all the models for 15 epochs and evaluate them on the validation set.
Figure 4.6: Variation of Accuracy of DIRECT and DIRE with length. The spike in DIRE’s performance for the last two categories with very few examples is likely to be an anomaly and not representative of its true performance on sequences of those lengths. Note that this is on the validation set.
### Table 4.3: Validation accuracy and Character Error Rate for advanced prediction versus first prediction. Both models are trained for 15 epochs. Refer to Section 4.2.2 for more details.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy (%)</th>
<th>CER</th>
</tr>
</thead>
<tbody>
<tr>
<td>First pred</td>
<td>34.4</td>
<td>.75</td>
</tr>
<tr>
<td>Advanced pred</td>
<td><strong>34.6</strong></td>
<td>.75</td>
</tr>
</tbody>
</table>

### Table 4.4: Validation accuracy and Character Error Rate for our encoder-decoder model versus a decoder-only model. Both models are trained for 15 epochs. Refer to Section 4.2.1 for more details.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy (%)</th>
<th>CER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoder Only</td>
<td>19.6</td>
<td>.97</td>
</tr>
<tr>
<td>Encoder-Decoder</td>
<td><strong>34.4</strong></td>
<td>.75</td>
</tr>
</tbody>
</table>

**Encoder-Decoder Architecture**

Table 4.4 shows the performance of our encoder-decoder model vs a decoder-only model (a single transformer, operating as an autoregressive language model) using the prediction at the first occurrence of each variable. As we can see, the decoder-only model does significantly worse, which is expected since it has access only to a part of the function while making a prediction at the first instance of a variable.

**Advanced Prediction Algorithm**

Table 4.3 compares the results of advanced prediction with “first prediction”, i.e., taking the prediction at the first occurrence of a variable. We observe that advanced prediction improves the performance of our encoder-decoder model by a small amount.

This could be explained by our observation in Section 4.4.2 that the model seems to rely its earlier predictions while predicting the name of a particular variable. Later predictions of a variable refer to the value assigned at the first prediction, and so the prediction of a variable seldom changes from what was predicted at the first instance.
Identifier Token Coefficient

We compare the performance of three different token weighting schemes in the loss function - weighting all tokens uniformly, weighting according to ITC (as described in Section 4.2.3), and weighting the identifiers only while ignoring the rest of the tokens.

As seen in Table 4.2, ITC shows a 4.4% increase in accuracy relative to the uniform weighting scheme, without hyperparameter tuning of the coefficient. As expected the model that ignores the surrounding tokens in the loss function performs worse. This is because the model doesn’t effectively learn the context surrounding the identifiers, resulting in a decrease in accuracy by 0.7 percentage points.
4.5 Threats to Validity

**Internal.** We made no effort to inspect the contents of our benchmark dataset. If the dataset contains an unrealistic representation of source code, such as an unusually sized vocabulary or an unusual number of repeated patterns our train, validate, and test splits may not be meaningful. We also assume that the decompiler used to generate the input sequences produces a reasonably accurate reconstruction of the original source code and that the variable matching done in constructing the dataset is in fact correct. Of course, there is also always the possibility of implementation bugs. We were unable to train DIRECT to convergence so the results will differ with more training time, but they will only improve, not worsen.

**External.** Since we didn’t inspect the contents of the benchmark dataset, it is possible that the benchmark dataset doesn’t accurately reflect the true nature of the source code. If these projects use an unusually simple vocabulary, they may overfit or underfit and not generalize. Identifiers are also used in many different contexts specific to the project, and there is no guarantee that the context in which these variables were used in the benchmark dataset is representative of other contexts.
4.6 Conclusion and Future Work

The problem of variable name reconstruction poses certain challenges for traditional transformer-based models. Specifically, the variable length of the prediction target, the constraints imposed by code syntax, architecture limitations that make long sequences difficult, and the task-specific non-uniformity of token significance. In this work, we developed a series of solutions to address these issues, namely 1) an encoding/decoding scheme to handle arbitrary sub-token length prediction, 2) a specialized prediction algorithm, 3) a customized identifier token coefficient weighting scheme, and 4) a splitting and combining algorithm for standard transformers to handle sequences of arbitrary length. In addition to empirical studies evaluating the effectiveness of each of these techniques, we also combined them to create DIRECT, a practical open-sourced identifier renaming engine. We evaluated DIRECT using a standard benchmark dataset against the state of the art, DIRE [97], and found that DIRECT provides a 20% improvement. We hope that in addition to an open-sourced tool, this work functions as a roadmap for other researchers trying to solve the types of problems we encountered when adapting transformer-based models to code analysis tasks. Future work could leverage the Abstract Syntax Tree (AST) of each function, and employ new transformer architectures like XLNet [192] to avoid splitting up the input while handling longer sequences. Our approach might also improve the results of other code analysis tasks like type inference, function re-naming, docstring prediction, and function boundary identification.
Chapter 5: REINFOREST

5.1 Introduction

With the rise of powerful large language models (LLMs) since the original BERT [37] paper was published, machine learning applied to source code analysis tasks has become increasingly popular in the last few years. At the same time as large public repositories of code have become available [175], the potential of code-to-code search has gained new significance to aid in common software maintenance and development tasks like code migration [83], transpilation [203], code repair [110], bug detection [82], education [193], and refactoring [198].

In this section, we present a series of novel techniques that enhance LLMs for code-to-code search and related tasks. We introduce a new code-to-code search technique that encodes both source code and dynamic runtime information during training without needing to execute anything for inference from either the search corpus or the search query. Figure 5.1 shows the general premise. First, during training the model learns both static and dynamic similarity from the training corpus, maximizing the distance between dissimilar code and minimizing the distance between similar code. We generate a single dynamic similarity feature we call the Semantic Similarity Score (SSS) before training that involves executing as much of the training corpus as possible with the same inputs and comparing the outputs of functions that have the same input types. This is the only time code is executed. Then to put this model to use, the model is used to create a precomputed embedding of the search corpus, and finally at inference time, the model embeds the search query so a simple comparison to the precomputed embeddings yields the search results. Traditionally dynamic analysis techniques require executing code during training as well as executing the search corpus and search query during the search procedure. This leads to problems scaling and all of the environment-related issues that come with running code in practice [191]. In contrast, our
approach only incurs the overhead of executing training set code in a controlled environment, and once the model is distributed code never needs to be executed again. Notably, our approach still accommodates circumstances when parts or all of the training set are not executable.

To validate the effectiveness of our approach, we considered the problem of cross-language search in which the query and corpus are from different languages, as in COSAL [120]. We felt this would best demonstrate the effectiveness of our technique since we are most concerned with behavioral similarity and searching across syntactically different languages means the model cannot rely on source code similarity.

We conduct a series of ablation studies that demonstrate the ability of LLMs to infer dynamic behavior under a variety of circumstances even when inference is based solely on static information. These circumstances include multiple query and corpus languages, various underlying LLM models both when the models are open-sourced and proprietary, when dynamic runtime information is available and without runtime information, and finally varying the number of both positive and negative reference samples in training. Our evaluation shows that our technique performs considerably better than state-of-the-art cross-language search techniques by up to 44.7% on the benchmark Atcoder [130] dataset, which consists of Java and Python solutions corresponding to 361 problems from 83 programming contests, and maintains near or better performance with a variety of LLM architectures including CodeBERT[47] and OpenAI’s codex models [16]. even when the LLMs cannot be fine-tuned. It is worth noting that we find that a smaller fine-tuned LLM outperforms significantly larger proprietary LLM’s highlighting the value and importance of open-sourced models for the research community. Further, we find that as long as there is at least one positive reference and one negative reference available during training our technique performs effectively compared to benchmarks, allowing for sparser datasets as what might be found in practice [153] and demonstrating the importance of considering both similar and dissimilar reference samples.

We make the following contributions:

1. The first code-to-code search technique encodes dynamic runtime information during training
if __name__ == '__main__':
    data = json.load(open(sys.argv[1]))
    ...

(a) Overview of the training process

Training Corpus
→ Code Search Model
→ Code Execution

(b) Overview of the inference process

Search Corpus
→ Precomputed Corpus Embedding
→ Query Embedding
→ Search Results

Figure 5.1: Overview of the training and inference processes

without the need to execute any code from either the search corpus or the search query during inference.

2. The first code-to-code search technique that considers both similar and dissimilar reference samples during training.

3. A set of ablation studies that demonstrate the ability of LLMs to infer dynamic behavior during cross-language search even though the inference is only based on static information.

4. An evaluation that shows our technique remains effective across many model architectures.

5. Ablation studies showing even a single positive and single negative reference sample in training results in effective performance.

6. An evaluation showing well-crafted fine-tuned models outperform larger non-fine-tuned LLM models even with the largest LLMs.

7. An open-sourced implementation of our tool and training procedures called REINFOREST¹ that represents a new state-of-the-art in cross-language code search.

5.2 Background

5.2.1 Code Search

Code-to-code search can be defined broadly as taking a code sample (the query), searching through a corpus of other available code samples (the search corpus), and finding code clones

¹https://github.com/reinforest-team/REINFOREST
in the search corpus matching the query. Cross-language code-to-code search deals specifically with situations when the query and corpus are written in different languages. Consider a situation in which Jane the developer is tasked with writing code in a language she is relatively new to or a language that lacks supporting documentation. Rather than writing without a guide, with cross-language code-to-code search Jane can make a query with a reference sample from a language she knows well and look for similar samples in a corpus of the language she is not as comfortable with.

Su et al. [168] define four type of code clones:

- **Type 1**: Identical code fragments, except for variations in whitespace, comments, or formatting.
- **Type 2**: Syntactically identical code fragments, with differences in variable, constant, or function names.
- **Type 3**: Similar code fragments with modified statements, function calls, or control structures.
- **Type 4**: Semantically identical code fragments with the same functionality but different syntactic structures.

Traditionally, type four clones are the most difficult to detect as the source code may offer little if any useful context, and that is the type we are most concerned with in this work. If our technique successfully learns code behavior, we should be able to perform code searches even when the syntactic structures are from different languages.

Code clones arise from several factors, including copy-pasting, code reuse, and adherence to coding patterns, and code search techniques are essential tools for developers, as they help locate and reuse code snippets in large codebases. Existing static code search techniques can broadly be classified into three categories 1) text-based approaches: These methods rely on traditional information retrieval techniques such as keyword search or string matching to find relevant code snippets, 2) structural approaches: These methods leverage the syntactic and semantic structure
of source code, such as abstract syntax trees, to find matching or similar code fragments, and 3) machine learning-based approaches, which leverage both text and semantic structural encoding along with machine learning algorithms to find clones. Traditionally code search has been limited to query and corpus of code from the same language, but in recent years this has expanded to search code bases where the query and corpus may be of different languages [78, 187, 188] as code search algorithms get better at distilling semantic information from the source code text.

Additionally, techniques that include information from dynamic analysis like execution traces [167] or input-output pairs [168] leverage additional context to find code clones that were undetected by static analysis. This is especially important in type four clones, which only concern behavioral similarity.

On the one hand static code search techniques do not require code execution, making them highly scalable for searching large codebases processing vast amounts of source code quickly, and providing fast search results to developers [151]. They are also easy to implement as text-based and structural approaches in static code search are generally simpler to implement than dynamic methods, and require no execution environment nor runtime setup making them more portable and applicable to a wide range of software projects. These practical benefits come at the cost of less context than what is available to dynamic techniques, which results in lower performance. On the other hand, dynamic techniques offer improved accuracy and context awareness but can be more complex and less scalable [116].

By including dynamic information during training where possible, and not requiring it at inference time, we aim to bridge the gap between dynamic and static code search bringing together both the convenience of static methods and the enhanced context of dynamic methods.

5.2.2 Encoder Models for source code understanding

With the advancement of Deep Learning models and techniques, models have been trained to "understand" source code. Given a code snippet \( c \) and an encoder \( Enc \), the model generates a \( d \)-dimensional vector representation of the code \( Enc(c) \in \mathbb{R}^d \). This vector represents the lexicographic,
syntactic, and semantic information in the code. Such vector embedding can be subsequently used for various tasks, including, but not limited to, classification, search, ranking, etc. The encoder can be an open-source model, such as CodeBERT [47] or a closed-source embedding API such as OpenAI’s codex embedding [140]. For a highly effective pre-trained model, the embeddings themselves should be sufficient for tasks related to code understanding however, as previous research has shown [135], the embeddings need to be tuned depending on the downstream tasks. In the case of open-source models for embedding, where we can access the models’ weights, the models can be fine-tuned as part of the downstream tasks. For embeddings coming from closed-source models, further models can be built on top of the initial embeddings, but the embeddings cannot be changed.

5.3 REINFOREST

In this section, we describe the code search technique we propose in REINFOREST. We describe REINFOREST as two disjoint steps – (a) Training and (b) Online query embedding and search. At the code level, REINFOREST consists of two different encoders – the query code encoder ($E_q$) and the document code encoder ($E_d$). Each of these, given an input code $x$, generates a fixed-sized vector representation of $x$. 

**Figure 5.2:** Training process in REINFOREST
5.3.1 Training REINFOREST

We train REINFOREST aiming to optimize for two different orthogonal goals. First, we would want REINFOREST to transform embeddings generated by the query code encoder \( E_q \) and the document code encoder \( E_d \) in such a way that minimizes the distance between related code and maximizes the distance between unrelated code. Our second goal is to endow REINFOREST with dynamic code similarity information during training so that, during inference (search), we do not have to calculate the semantic (I/O) similarity between the query code and all other code in the database. Figure 5.2 shows an overview of REINFOREST’s training process.

**Contrastive Training.** To achieve the first goal, we take inspiration from contrastive learning [24]. We assume the presence of a dataset \( D \) consisting of a set of tuples \((c_s, C_p, C_n)\). \( c_s \) is a piece of code written in the source language \( s \) (e.g., Java). \( C_p \) and \( C_n \) are the set of code written in the target language (e.g., Python). The set \( C_p = \{p_{t1}, p_{t2}, ..., p_{tk}\} \) are the solutions to the same problem as \( c_s \). In contrast, \( C_n = \{n_{t1}, n_{t2}, ..., n_{tk}\} \) are solutions to a different problem than \( c_s \). We refer to the code samples \( C_p \) as positive samples and code samples \( C_n \) as negative samples hereafter. Intuitively, we would want to bring the embeddings of each element in \( C_p \) closer to the embedding of \( c_s \), and we would want to push the embedding of \( C_n \)’s elements far away from \( c_s \)’s embedding.

First, we encode \( c_s \) with the query code encoder, \( E_q \), and both the \( C_p \) and \( C_n \) sets with the document code encoder \( E_d \). Formally, \( R_s = E_q(c_s) \), \( R_p = \{E_d(p_{ti}) \mid p_{ti} \in C_p\} \), and \( R_n = \{E_d(n_{ti}) \mid n_{ti} \in C_n\} \), where \( R_s \) is the embedding of the query code, \( R_p \), and \( R_n \) are the sets of embedding corresponding to positive and negative samples, respectively. Then we compute the cosine similarity between the query code and each of the positive and negative samples. We compute the cosine similarity between two vectors, \( a \) and \( b \) as,

\[
\text{sim}(a, b) = \frac{|a \cdot b|}{|a| \cdot |b|} \tag{5.1}
\]

Our next goal is to simultaneously maximize the similarity between \( c_s \) and all elements in \( C_p \) and minimize the similarity between \( c_s \) and all elements in \( C_n \). We achieve such an objective by
minimizing the following loss function,

\[
L = \sum_{(c_s, c_p, c_n) \in D} \left( \sum_{r_{pi}^p \in R_p} (l_p - \text{sim} (R_s, r_{pi}^p))^2 \right) + \left( \sum_{r_{ni}^n \in R_n} (l_n - \text{sim} (R_s, r_{ni}^n))^2 \right) \tag{5.2}
\]

Here, \(l_p\) and \(l_n\) are target labels for positive and negative samples. The values of \(l_p\) and \(l_n\) should depend on the similarity function being used in the system. For an unbounded similarity function, \(l_p\) and \(l_n\) should be set to \(\infty\) and \(-\infty\), respectively. In the case of REINFOREST, we set \(l_p = 1\) and \(l_n = 0\), since these values are the maxima and minima of the similarity function we use (Equation (5.1)).

**Semantic Similarity Score** As our second training goal is to endow the models with the knowledge about semantic (I/O) similarity, we assume a function \(S_{io}(c_s, c_t)\), which computes the semantic similarity between \(c_s\) and \(c_t\). We endow the model with such semantic similarity by modifying the loss function in Equation (5.2) as

\[
L' = \sum_{(c_s, c_p, c_n) \in D} \left( \sum_{p_{pi}^p} (l'_p(c_s, p_{ti}) - \text{sim} (R_s, r_{pi}^p))^2 \right) + \left( \sum_{n_{ti} r_{ni}^n} (l'_n(c_s, n_{ti}) - \text{sim} (R_s, r_{ni}^n))^2 \right) \tag{5.3}
\]

\[
l'_p(c_s, p_{ti}) = (1 - \alpha)l_p + \alpha S_{io}(c_s, p_{ti}) \tag{5.4}
\]

\[
l'_n(c_s, n_{ti}) = (1 - \alpha)l_n + \alpha S_{io}(c_s, n_{ti}) \tag{5.5}
\]

Here, \(\alpha\) represents the relative importance of semantic training. In an ideal scenario, where we could compute the semantic similarity between all pairs of \(c_s\) and \(c_t\), we argue that setting \(\alpha\) to 1.0 would result in the best performance. However, it is practically infeasible to assume that for all such pairs of \(c_s\) and \(c_t\), \(S_{io}(c_s, c_t)\) exists. Thus, we empirically found the best validation performance by setting \(\alpha\) to 0.2. Regardless, \(\alpha\) remains a user-defined hyperparameter in REINFOREST’s implementation.

**Semantic Similarity Score Calculation For Training.** REINFOREST’s architecture and process uses a semantic similarity score calculation based on runtime information where possible during training. We use SLACC’s similarity scoring methodology, which is simply the number of matching outputs divided by the number of inputs. Figure 5.3 outlines our two process procedures
for generating similarity scores. First, as shown in Figure 5.3a, we generate the input corpus from the coded corpus. For any piece of code, we statically analyze the system calls and derive the structure of the input for that particular piece of code. After extracting all of the input structures in the corpus, we generate a large number of inputs of random values for each structure based on primitive types and create an input corpus. Then as shown in 5.3b for any two samples, we extract the input structures, run them with the inputs that match their input structure, and then perform the similarity calculation described earlier. As in SLACC, if two pieces of code have different input structures, they receive a similarity score of zero.

5.3.2 Code Search With REINFOREST

Once the training process finishes, we have two trained encoders, $E_q$ and $E_d$, to encode queries and documents, respectively. Immediately after the training, we encode all the code in the search database with $E_d$. Since the encoded representations are real-valued vectors, indexing tools such as FAISS [75] can store them in the indexed database for efficient search. For every query code $c_s$, we
generate the query embedding $R_s$ with the query encoder $E_s$. Then we search in the embedding database based on the similarity score in Equation (5.1) and return the top $n$ candidate with the highest similarity scores.

5.4 Experimental Design

5.4.1 Research Questions

To evaluate our technique we implemented it in a tool called REINFOREST and we considered four main questions. The most important and broadest of which is simply:

**RQ1. How does REINFOREST’s performance compare to the performance of other cross-language code search techniques?**

Still, better performance could mean that we simply rely on the power of modern LLMs. If, however, our tool continues to show state-of-the-art performance with multiple LLMs, and our additional training method always improves performance this demonstrates that our technique is in fact the source of any performance advantages shown in RQ1. So we ask:

**RQ2. Does REINFOREST’s methodology and performance generalize across different models?**

Even if RQ2 shows performance improvements across multiple architectures, it may only be from source code analysis and the dynamic runtime information may not make a difference, so we ask:

**RQ3. Does including semantic similarity scores during training improve code search?**

Lastly, it remains unclear what impact including both positive and negative samples during training has, so we ask:

**RQ4. How does changing the number of positive and negative comparison samples available for training effect REINFOREST’s performance?**

5.4.2 Dataset

To evaluate REINFOREST, we used the Atcoder dataset [130], which consists of 18644 Java solutions and 22317 Python solutions corresponding to 361 programming contest problems. To
Table 5.1: Statistics of the dataset. The “Average” denotes the average number of files per problem.

<table>
<thead>
<tr>
<th></th>
<th>Train</th>
<th>Valid</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Problems</td>
<td>287</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>Number of Java files</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>14413</td>
<td>2533</td>
<td>1698</td>
</tr>
<tr>
<td>Average</td>
<td>50.22</td>
<td>68.50</td>
<td>48.89</td>
</tr>
<tr>
<td>Number of Python files</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>17182</td>
<td>2983</td>
<td>2152</td>
</tr>
<tr>
<td>Average</td>
<td>59.87</td>
<td>80.62</td>
<td>58.16</td>
</tr>
</tbody>
</table>

prove our concept, we created the train, validation, and test splits by dividing the dataset across different problems – *i.e.*, train, valid, and test splits do not share the solution from the same problem. Since the problems are independent of each other, dividing in such a way prohibits data leakage across splits. Across all the experiments presented in the paper, we evaluate based on the same splits. Table 5.1 shows detailed statistics of the dataset. All experiments were performed with an A100 NVIDIA with 216 GB of RAM with a 2.44 GHz AMD 64-core processor.

5.4.3 Evaluation Metrics

we used three different evaluation metrics. The first is precision at N (PR@N), where N is between one and five inclusively. Precision at N is calculated as the average number of truly positive examples in the first N search results using every sample in the test set as a query against the rest of the test set. These metrics are benchmark standards in code search [13], but we feel they are most important as they are indicative of the experience a user would have looking at a list of search results following a query. We also examined a metric termed Average Rank Gap (ARG) [128], which is defined as the average rank of negative examples minus the average rank of positive examples. Unlike PR@N metrics, the ARG gives a sense of the entire set of search results available in the corpus. Between the PR@N metric and the ARG metric, we get a sense of what a user would experience as well as REINFOREST’s performance across the entire dataset. Lastly, we evaluated the tools based on the average first position (AFP) [184] of a positive result (lower is better). ARG provides context for the entire result set while AFP provides context for the highest-ranked portion of the result set as a whole.
5.5 Empirical Results

We answer the questions in the previous section one by one here. A full table of our results has been omitted for space consideration and clarity but is included in the supplementary material.

5.5.1 RQ1 - Overall Performance

The first thing we examined was the overall performance of the model, i.e., how good is the model at finding cross-language clones. Using the evaluation metrics defined in the previous section, we evaluated both techniques that require training as well as those that do not. For strawman comparison, we implemented a TF-IDF search with the BM25 python library [114] that treats source code like natural language and performs a token-based search. We also implemented an AST-based search adapted from SLACC [119], in which we take an AST representation of both the query and the document in the corpus, and convert them from language-specific ASTs to a generic AST with common node types where possible. Then we use the standard tree distance algorithm [201] to determine the similarity of the two samples. In the last of our non-training comparisons, we reimplemented COSAL’s [120] token subset technique in which functions are divided into smaller snippets, and the smaller snippets are clustered to determine a similarity between functions. We reimplemented this system because it is state-of-the-art in cross-language code search, and we have different train, validate, and test splits on the Atcoder dataset compared to their Atcoder experiments.
and we have a different experimental setup.

To compare against other code search techniques that require training, we selected three state-of-the-art LLM models that have shown promising results in a variety of code-related tasks. CodeBERT [47] is a pre-trained model for code representation and generation developed by Microsoft Research Asia. It can be fine-tuned for various downstream tasks and has achieved state-of-the-art performance on several benchmark datasets. CodeBERT’s ability to transfer knowledge across programming languages makes it particularly useful for code generation tasks and has many potential applications in software engineering and programming language research. GraphCodeBERT [47] is an extension of CodeBERT that incorporates control flow graphs (CFGs) into its pre-training process. By using CFGs, GraphCodeBERT can better capture the relationships between code tokens and produce more accurate representations for code analysis and generation tasks. It has achieved state-of-the-art performance on several code analysis benchmarks, demonstrating its effectiveness in modeling complex code structures. UnixCoder [56] is a pre-trained model for natural language to shell command translation, developed by Facebook AI. It is trained on a large corpus of command-line usage examples and can generate shell commands given a natural language input. UnixCoder has achieved state-of-the-art performance on several benchmarks, demonstrating its ability to understand and translate complex natural language commands into executable shell commands.

We implemented a search algorithm with each of the three models as described in Figure 5.4, similar to that of [112]. The models are trained as a classifier on the training set and then for every document in the test set the classifier determines the probability of the document matching the given query and sorts results based on the highest probability. As such at inference time this process requires classifying the entire test corpus and incurs tremendous computational overhead. **ReINFOREST** in contrast can use a precomputed corpus embedding as shown in Figure 5.1b and only needs to create an embedding vector from the query without concerning the rest of the corpus. To get the search results, simply choose the precomputed embeddings that are closest to the query vector. Performance experiments for our best version of **ReINFOREST** took roughly 14 hours while the other trained model code search comparisons would have taken multiple days to complete. To
deal with this we randomly sampled 200 queries from the test set and reported our findings for those 200 samples. With this stratified sampling, the experiments took roughly the same amount of time as the experiments with REINFOREST.

Figure 5.5 details the results of our experiments for PR@N metrics and ARG with the best-performing version of REINFOREST against all the comparison techniques. For visual clarity, PR@N metrics were normalized to a percentage scale, ARG metrics were scaled by 100x, and AFP metrics were omitted but original values are included in the supplemental material. Figure 5.5a shows results for a Java query against a corpus of Python code while Figure 5.5b shows results for a Python query against a corpus of Java code. Across all evaluation metrics REINFOREST outperforms all comparison techniques with improved performance up to 40% compared to the next best performing method. Somewhat surprisingly we see the no training required techniques outperforming the trained techniques with the exception of UnixCoder and Java to Python search. While the relative performance of each method changes depending on the evaluation criteria, we find unsurprisingly that current state-of-the-art COSAL generally is the next best performing.

Since REINFOREST outperforms existing cross-language code search techniques by a considerable margin on all evaluation metrics we determine the answer to RQ1 as:

**Result 1:** REINFOREST performs considerably better than existing techniques with a 40.6% improvement over the nearest baseline for Java to Python search, and an 11.5% improvement over the nearest baseline in Python to Java search. We outperform COSAL’s state-of-the-art performance by up to 44.7%.
5.5.2 RQ2 – Model Generality

While the results from the previous section show that the techniques described result in an effective tool, it remains unclear if the performance is because of the training techniques we employed or just the power of modern LLMs. To examine this we ask in RQ2 if our REINFOREST techniques are effective in improving performance across many different LLM models both open-sourced and proprietary.

For performance comparison, we use the same metrics as described in the last subsection: PR@N, ARG, and AFP. We chose the most advanced available LLM models for comparison. First CodeBERT for the reasons described previously, but instead of training it as a classifier as we did in the first RQ, we fine-tuned its embedding on the cross-language code search task during the training procedure described in Section 5.4. Additionally, we ran our training models on OpenAI’s Codex Ada, Babbage, Curie, and Davinci models. These are proprietary models and as such we could not perform fine-tuning during training, so we used the encodings from these models as is during training and REINFOREST learns to maximize and minimize distances in these encoding spaces. Without access to the model internals, the only discernible difference between the OpenAI models is the vector size of the embeddings, which are Ada: 1024, Babbage: 2048, Curie: 4096, and Davinci: 12188. As such we cannot explain why some OpenAI models may perform better than others.

To answer this question we ran two experiments. First, we compared the performance of the various models’ embeddings on cross-language code-to-code search without any alterations to REINFOREST’s performance on the same task training on the embeddings from those same models. Any increase in performance would have to be attributed to REINFOREST’s procedures instead of the base LLM.

Second, we compared the performance of REINFOREST trained with the various underlying model embeddings on each evaluation metric to the next best-performing code-to-code search technique. To understand the significance of the generality of our technique we should examine it relative to the state-of-the-art across all model architectures. Achieving state-of-the-art performance with all underlying models would further suggest that our techniques are valuable independent of
the underlying LLMs.

Figure 5.6 compares performance between trained and untrained models. Clearly REINFOREST’s training always improves code-to-code search performance for both Java to Python and Python to Java queries. The improvements range from 5.2% all the way up to 1044%. Unsurprisingly our training has the largest impact on the one model that we could fine-tune during the training procedure. More interestingly, however, we find that despite having the weakest performance without training the fine-tuned CodeBERT model outperforms all other models on both Java and Python cross-language queries. This shows that fine-tuning the embedding during the training procedure has a larger impact than the size of the overall LLM and demonstrates the importance of fine-tuning.

Figure 5.7 shows the overall performance of the various models using REINFOREST’s techniques relative to COSAL (SOTA). Figure 5.7a details the results of Java to Python queries and shows that REINFOREST achieves SoA performance across all metrics. Interestingly the same cannot be
said for Python to Java queries as COSAL outperforms REINFOREST in some circumstances but never outperforms all versions of REINFOREST. Since we do not know the internals of OpenAI’s models it remains unclear why some of their models perform better than the SoA while others do not. However, most importantly we find that fine-tuned CodeBERT outperforms all other models on every metric for both cross-language queries. This solidifies fine-tuned CodeBERT as the highest-performing version of REINFOREST and demonstrates the power of open-sourced models. It also suggests that if the proprietary models were open-sourced we might see greater improvement.

The results from these experiments show that 1) REINFOREST’s training techniques always improve the base LLM models and 2) REINFOREST’s techniques outperform the state-of-the-art in most circumstances and always when fine-tuning is possible. Thus we find that:

**Result 2:** REINFOREST’s techniques generalize across multiple LLM models, improving performance by a minimum of 26.2% on Java to Python search and 5.17% on Python to Java search, and when the base LLM is fine-tuned during training performance improves by 7.69x on Java to Python search and 9.96x on Python to Java search.

5.5.3 RQ3 – Impact of SSS

While the results from the previous RQ show that REINFOREST’s techniques are in fact responsible for the performance we saw in RQ1, we don’t know how much of this performance is caused by the inclusion of runtime data in the form of the semantic similarity score (SSS). The SSS is the scored input-output comparison described in Section 5.4.

To answer this question we compare the performance of trained versions of REINFOREST when the SSS is included in the training procedure and when it is not. To perform these experiments we simply withheld the SSS from the training procedure. Any performance improvement would have to be from the model learning runtime behavior from the SSS. We tested this with CodeBERT as well as the OpenAI models to see if learning runtime behavior is in fact an inherent property of modern LLM’s or peculiar to a single model.

Figure 5.8 shows the impact of SSS on REINFOREST’s performance with various underlying
models and cross-language queries. We find that including the SSS, regardless of the query language or the model used for embedding, always improves performance. This means that the models are in fact learning dynamic behavior despite never running the code during inference. Further, it shows that this capability is inherent to modern LLMs and is not a property of a single model. We also see that it has the most improvement on one of OpenAI’s proprietary models (davinci) which means that this technique continues to improve performance regardless of whether the model is fine-tuned during training or not.

Thus we conclude that

**Result 3:** Including SSS improves REINFOREST’s code search performance across all embeddings. Including the SSS with the best-performing version of REINFOREST(CodeBERT) contributed to a 7% improvement for Java to Python queries and a 4.8% improvement for Python to Java queries.

5.5.4 RQ4 – Varying Training Samples

Previous experiments show that 1) REINFOREST’s overall performance is better than the state-of-the-art, 2) is consistent regardless of the underlying model architecture, and 3) leverages runtime behavioral features. However, it remains unclear if the model is learning from both positive and negative reference samples during training or if it is simply relying on one or the other. Further, we have not yet investigated the impact of each reference sample on the model’s performance. It seems plausible that increasing the size of the training set or adding more samples could improve performance.
So we investigate RQ4 by varying the maximum number of positive and negative samples available during training. If the model is only learning from positive samples then performance should not dip when negative samples are removed and vice versa. Further, if more training data would improve performance then if the number of reference samples available during training increases performance should increase as well. We ran performance experiments with the same number of positive and negative reference samples for one, three, and five references. Then we ran the experiments with zero positive references and five negative references and vice versa. we did this with a CodeBERT model fine-tuned during training and all available OpenAI Codex models off the shelf as in RQ2.

The full results of these experiments are included in the supplementary materials with all models and evaluation metrics. Figure 5.9 shows these results with both Java to Python search and Python to Java search for CodeBERT and OpenAI’s Ada model evaluated on the PR@1 metric. The x and y axes show the number of positive and negative samples available during training and the size of the circles represents the PR@1 performance of the model in the circumstances described by the x and y axis. Larger is better.

There are two interesting findings consistent in both models that can be fine-tuned and those that can not. The first is that when either positive or negative samples are omitted during training, performance decreases drastically. This indicates that REINFOREST’s dual maximizing and minimizing technique during training is in fact responsible for its performance. If only the positive samples were responsible for performance, then removing the negative samples would have no impact. This
is clearly not the case. Second, we see that performance doesn’t vary greatly when a single positive and negative reference sample is available. This means that increasing the size of the training set would not have a meaningful impact on results. Still, we do see a slight increase indicating it has some impact. This could be explained by the fact that during training source code labels are binary, which means that additional positive and negative samples do not explicitly provide extra context. With a different training metric that quantified the distance between samples more explicitly it is possible that the number of samples available would have more impact. However, this finding indicates that even in the case of a sparse dataset available during training, REINFOREST’s performance is still effective even if only a single positive and negative reference sample is available.

**Result 4:** REINFOREST’s performance depends on both positive and negative samples. Using a combination of positive and negative samples improves performance by up to 10.2x and 17.8x for Python to Java search and 15.5x and 12.2x for Java to Python search, compared to only positive and negative references respectively. The addition of more than one positive and negative sample during training performance improvement decays, with a 5.4% and 0.6% improvement for Java to Python and Python to Java search, respectively.

5.6 Threats to Validity and Limitations

5.6.1 Internal

Since we had unique training, validation, and testing splits forming a singular experimental setting compared to results reported by the benchmark comparison techniques including COSAL, we were forced to implement the benchmark techniques ourselves as best we could. While the NLP-based search and the AST diff algorithm are fairly straightforward, recreating COSAL’s snippet execution technique is somewhat complex. It is possible that some errors were made in recreating their results but the results in our evaluation are consistent with their findings. We evaluated benchmark techniques that involved training based on a stratified sampling of 200 randomly selected queries from the test set. It is possible that performance would differ when running the trained code search benchmarks over the entire dataset.
We evaluated the Atcoder dataset in order to compare it with COSAL’s state-of-the-art performance, and in order to compile the SSS as many samples as possible were executed but we were unable to execute much of the dataset since it did not come with input specifications. When an SSS could not be collected we used a default value. It is possible that SSS could have a bigger or smaller impact if we had been able to generate inputs for the entire dataset. Furthermore, during SSS collection there were some sample pairs that were labeled as true positives but had different input types since the dataset allowed identical solutions with equivalent but different input structures (i.e. set vs list). These behavioral similar pairs scored an SSS of 0 during training and finding a true SSS for these pairs may change results.

We found that setting $\alpha$ to 0.2 had the best results of the values we tried, but more rigorous hyperparameter tuning could result in a different optimal value.

5.6.2 External

The Atcoder dataset is compiled from submissions to a coding contest by practitioners of all levels. It remains unclear if the code in these contests is actually reflective of production code. Further, we examine code-to-code search when two functions are defined to solve the exact same problem. It is unlikely to find such examples in practice and it remains unclear how well REINFOREST would perform in real-world scenarios. As was mentioned earlier 5.4 we only consider behavioral clones, or not, during training and this limits the amount of context the model can learn from multiple positive and negative samples. Our experiments show the presence of both positive and negative samples impact performance but with a different training procedure that considered the distance between samples rather than classifying them as one of two categories the addition of more available samples during training may have more impact on performance.

5.6.3 Construct

We chose our evaluation metrics based on what we believed to be standard and meaningful. There are other important evaluation metrics like recall, overall accuracy, and F1 score that we did
not measure. We chose cross-language search because we believe that it was the most meaningful problem in identifying behavioral similarity, but experiments on same-language search would need to be carried out explicitly to evaluate ReINFOREST’s effectiveness for that purpose.

5.6.4 Limitations

Finding appropriate inputs that can be used to calculate the SSS between two pairs of functions is analogous to input (test case) generation which remains an open research problem. Further, our method depends on a single function-to-function mapping which may not occur in practice, as the same behavior can be divided between multiple functions in different codebases.

5.7 Conclusion

This chapter presents a novel code-to-code search technique and an implementation of that technique called ReINFOREST. The technique leverages modern LLMs and enhances them by encoding runtime behavior in the form of a Semantic Similarity Score (SSS). Unlike other code search techniques that only consider positive samples during training, ReINFOREST both minimizes the distance between similar samples and maximizes the distance between dissimilar samples. We evaluated ReINFOREST on cross-language code search, both searching a corpus of Python samples with Java queries and vice versa. We found that our technique outperforms the state of the art on all evaluation metrics. Further, we found that our training technique always improves performance over modern LLM encodings out of the box, regardless of the underlying model, and we found that ReINFOREST performs especially well when the LLM models can be fine-tuned. This proves that our procedures presented here are the source of the improved performance and not the underlying models. We also present experiments quantifying both the impact of SSS and the importance of both positive and negative samples available during training. We found that including the SSS always improves performance, indicating the model does in fact learn dynamic behavior without dynamic context available at inference time, and that performance depends on the presence of both positive and negative samples. The complete set of results including results that were omitted for
clarity is included in the supplementary material and all models and training procedures will be open-sourced.
Chapter 6: Binary Patch Decomposition
6.1 Introduction

In most cases, the typical software maintenance lifecycle is sufficient to keep code updated and secure. Users report bugs, developers investigate the issue, resolve the problem, release a new version of the software in line with the pre-existing release schedule, and users deploy the new version of the software. However, in software that deals with critical system functionality users tend to update software only when absolutely necessary for fear of updates introducing side effects that disrupt service. Since most software is released according to a common schedule, new releases often contain many modifications most of which a singular user would deem unnecessary. These pose an unnecessary risk and in fact, the user may not be able to integrate new versions if relevant interfaces have been replaced or wiped away. This leaves many users in an awkward position: they have code with known deficiencies and the corresponding updates, but they also can’t apply those updates.

While not necessarily problematic for updates concerning standard feature releases or enhancements, this situation proves disastrous when failing to fix security vulnerabilities. In the Equifax hack of 2017, [134], attackers exploited a well-known vulnerability in the Apache Struts library that had been fixed months earlier but had not been applied to Equifax’s production code, to steal approximately 145.5 million U.S. consumers’ personal data, including their full names, Social Security numbers, birth dates, addresses, and driver license numbers.

If users had access to the version control repository source code and build process, they could theoretically search the change log to build the specific version that suits their needs, but proprietary and legacy code users either don’t have access or no such build exists since many unrelated changes have been included in the update. Instead, we present a new technique called binary quilting that allows users to apply the minimum patch designated by the developers as fixing only the targeted software bug. This eliminates side effects from unwanted changes, in the new release, thus providing a way for users to apply updates to legacy binaries while still supporting necessary functionality.

Since there are many different ways to take the diff between two binaries, we developed a
technique that integrates with the build process called Binary Patch Decomposition (BPD) that automatically creates the necessary metadata for the quilting procedure (ATTUNE). BPD creates metadata associated with each commit that might be of interest to users in the context of a specific build that adds only that commit to the prior release (the buggy release in the production user environment). The developers then supply this metadata for each bug-fix patch along with the new release (potentially consisting of many commits), which the developers must make available to users for ATTUNE to operate in the user environment while still concealing source-level semantics and preserving source code security. We only require that the developers make this metadata available to the users.

This work presents the following contributions:

1. **Binary Quilting** - a technique to generate patched binaries by combining executables. To try to minimize confusion, we henceforth refer to Egalito’s notion of “recompiling”, from its intermediate representation without source code or a conventional compiler, as “(re)generation”.

2. **Binary Patch Decomposition** (BPD) - a build process integrated technique to map specific source code changes to the corresponding binary changes.

3. ATTUNE’s implementation of BPD and quilting for Linux on x86.
4. Case studies demonstrating ATTUNE’s potential in realistic update scenarios.
6.2 Background

6.2.1 ELF Format and Existing Binary Diffs

ELF files are the standard executable format on Linux operating systems, which contain all the metadata required to link, load, and run the program. This extra information includes the symbol table which maps the location of every symbol in the binary to the name of the symbol in the associated string table. The string table is simply a list of strings delimited by null bytes. This data structure allows for linking across object files and executable ELF files. The ELF file also contains a relocation table for link time and load time symbol resolutions. In many production binaries symbols are stripped for performance and security so may not be available for analysis. Lastly, the ELF format splits the binary into data and code segments with different permissions for security reasons.

Binary diff is a well-studied problem, however, it is usually taken from the perspective of a reverse engineer. Such tools include BinDiff [205], BinHunt [51], iBinHunt [125], BinSlayer [14], BinSequence [71], and SemDiff [179]. All of these tools rely on some variant of control flow graph extraction and comparison. Usually this entails significant computing overhead and heavy analysis. However, in the context of binary quilting, these techniques fail to leverage the information available during the build process, so these diffs may pick up on changes not relevant to the patch desired by a given user, resulting in an inaccurate quilting effort.

6.2.2 x86 Calling Conventions

While the minimum binary diff information provided by BPD relays where changes occur, since small changes in source code can result in large semantic changes like changing register usage and memory layout, copying semantically equivalent or nearly equivalent code from one binary to another is precarious. Fortunately, the Intel x86 architecture defines strict calling conventions as to what registers must be maintained by the calling and callee functions as well as the state of the stack for arguments to be passed from one function to another [49]. As long as the arguments are
consistent between the two function versions, the initial state of the stack and the register layout are presumed to be the same. In the event a function’s arguments change, this guarantee no longer holds true, but the calling functions must also change to accommodate the new signature so for the purposes of applying patches, these calling functions provide the points of interface between original and new versions.
6.3 Design

6.3.1 BPD Design

Source code diff analysis. Standard version control systems (VCS) like Git use source code diffs to track source code changes. For any given code change the VCS maintains a record of exactly what changed. Most VCSs save this record following a standard patch file convention as shown in Figure 6.2. This patch file includes the modified function name, line numbers, source code changes, and file names among other information.

Symbol Table Parsing BPD parses the patch file and extracts the names of the modified function(s). In the event that the change modifies a data-related symbol like a global variable or a string then that symbol is marked as modified. After the executable is built ATTUNE’s BPD parses the symbol table to extract the location of the modified function in the new binary. The symbol name provides an interface between the source code changes and the binary addresses as shown in Figure 6.3 where the addresses are denoted in hex on the left and the symbols in text on the right. The symbol table also includes a description of the data type. In the example shown the symbols are function names designated by \texttt{FUNC} but if they were data symbols like global variables or structs they would be designated \texttt{OBJECT}. This information is relevant because depending on the data type they will be in different sections of the binary. The symbol table also specifies the size of the symbols so we can extract the length of the functions and the size of the data types. Once we know the location, size, and contents of the modified, even if the symbol table is stripped after building (usually pre-deployment) we can still identify the patched portions of the binary completely.

Database storage for on-the-fly patch decomposition BPD is designed to track binary changes without interfering with the normal development process. One way to track changes would be to keep a separate copy of the binary for each commit and then extract information as needed to create the binary patch. However, keeping so many versions of the binary doesn’t really make sense since most of the information is redundant. BPD stores only the necessary information to construct the
... @ -3167,10 +3167,13 @@ png_check_chunk_length(...) {
...
- (png_ptr->width * png_ptr->channels // source changes
+ (size_t)png_ptr->width
+ * (size_t)png_ptr->channels

Figure 6.2: libpng-bug-1 Abbreviated Example Patchfile

0000000000003fe0 56 FUNC ... png_check_chunk_name
0000000000004020 221 FUNC ... png_check_chunk_length
0000000000004100 172 FUNC ... png_read_chunk_header

Figure 6.3: libpng-bug-1 Abbreviated Symbol Table Entries

---

Algorithm 4: Metadata Construction Algorithm

**Result:** Metadata(old_info, new_info)

**Input:** parsed original binary OV; parsed new binary NV; BPD datastore DB

**Output:** metadata information required to construct patch MD

`getMetadata (OV, NV, DB)`

original_code = DB.get_code_pieces(OV);
new_code = DB.get_code_pieces(NV);
res.new_info ← ∅, res.old_info ← ∅;
changed_symbols = DB.getChangedSymbols(NV);

**foreach** symbol ∈ changed_symbols do
res.new_info.add(symbol);
if symbol ∈ original_code then
res.old_info.add(symbol);
**foreach** cp ∈ symbol.dependency_list do
if cp ∉ original_code then
sym = Symbol(cp, newChange=True);
res.new_code.add(sym);
end
end

return Metadata(res.new_info, res.old_info);
binary patch as shown in Figure 6.4. Essentially the binary is split based on its symbols and the symbol contents and metadata are stored in the database. The database must reflect the binary’s structural information to apply the patch after symbols have been stripped, its contents so the patch can be built, and maintain flexibility so a patch can be constructed from multiple code versions. This means that each piece of code must keep track of its external references across versions.

**Metadata Construction Algorithm** When a user asks for a particular patch, **ATTUNE**’s BPD must transform the entries in the database into metadata that can be leveraged to build and apply the patch in the user environment. The process to construct this metadata is described in Algorithm 4. The algorithm takes the parsed original and new binaries as inputs, and then for every changed symbol it has to do two things. First, if the symbol exists in the old binary, it must add the old size and position data so any references to this piece can be removed. Second, BPD must search through all the dependencies of each changed symbol (a dependency is any nonlocal reference); if the dependency existed in the old binary (as per the symbol name), then the metadata can simply add the new code piece, and the location of its own dependencies. If the dependency doesn’t exist, it must be added to the metadata as a newly changed symbol and its dependencies searched as well.

Since BPD keeps a record of all changes the dependencies are on versions of code not just symbol names, so even though the required symbol may appear in the original binary, if the symbol has been updated between versions BPD includes the metadata for the most recent one.
Figure 6.4: Datastore Implementation
Algorithm 5: Pseudocode: Binary Quilting

**Result:** Updated Pointers

**Input**: parsed original binary OV; parsed new binary NV; BPD metadata MD

**Output**: The quilted binary

`getQuilted(OV, NV, MD)`

```plaintext
foreach ref ∈ OV do
    if isOverwritten(ref.target) then
        ref.target = metadata.new_info[target_symbol];
    end

decase ref ∈ metadata.ne_info do
    if isCodeRef(ref.target) then
        // adjust pointer
    if isDataRef(ref.target) then
        if OV.dataSection(ref) then
            // adjust pointer
        else
            // add new data
            // adjust pointer
        end
    if isPLTRef(ref.target) then
        // update PLT entries
    "end
```

6.3.2 Binary Quilting Procedure

Once the BPD metadata has been produced and shipped to the user as described in the previous section, the user must run ATTUNE’s quilting function to actually create the patched binary and the fully linked form of the patch. The fully quilted binary is shown in Figure 3.8.

In order to parse the binaries and identify the reference locations as well as the regeneration phase we rely on the Egalito framework. It lifts the binary into an intermediary representation
(EIR) that extracts the control flow graph and provides an interface to the data sections. Egalito also provides a portable XML format (HOBBIT) that encodes all the structural and data-related information. Egalito then generates an ELF file to run on the specified hardware. This provides some interesting opportunities for updating legacy binaries to different hardware.

In order to quilt the new version of the binary into the old version, we must handle three different types of references. These are code references, data references, and external references. The algorithm is described in Algorithm 5. Each link from the old binary to the old versions of the patched code must be removed, and updated to point to the new versions where appropriate. In the event that the new binary relies on an updated version of a library, the user must have access to this version. However as long as the new binary has the correct external references, then the quilting procedure still supports it. This becomes particularly useful in the context of dealing with legacy binaries since patching a legacy binary may rely on pieces of new library versions. This means ATTUNE’s quilting technique could support multiple versions of the same library as long as BPD provides the correct metadata.

**Code section quilting** One important change in newer binaries is that they are built with position-independent code (PIC) by default. PIC code became the default on Ubuntu 17.0 across all architectures in 2017 [100]. For security reasons PIC code is designed with relative references so can be deployed anywhere in the address space, but this also means it can be moved from one binary to another. Older code with absolute references on the other hand requires finding the corresponding reference in the new binary. *In order to do this the symbols become a point of reference between the binaries*. Each absolute reference can be represented as a symbol and offset. This symbol and offset remain constant across versions even though the absolute addresses will change across versions.

First the links from the original binary to the code that has since been updated need to be removed. The EIR is scanned for any references that point to dead code, and the links are deleted. In most cases, these are function pointers. The metadata contains the translation data to update the pointers from the old symbols to the appropriate symbols in the new binary. Next, the patched code from the new binary needs to point to the correct places in the original binary. In the event that the
patched code depends on code only in the new version these new versions of the code need to be quilted in as well. This allows the quilted binary to be a composite of multiple versions. ATTUNE can quilt all combinations of PIC and non-PIC code. The binary patch decomposition metadata contains both the locations of references in the new version that need to be linked, as well as where those links should point. We depend on Egalito to generate the corresponding ELF file.

**Data section quilting** Quilting data section references and code section references can be done through similar procedures. Remove links to code that have since been updated and add links to the new code. However, what happens in the binary is quite different since references to the data section are in different parts of the binary. The ELF specification splits the binary into data and code sections for security reasons. This introduces some complications to quilting. The code references pointers in the global offset table (GOT) which point to the correct part of the data segment. The second difficulty comes from having to reformat the data segment if it needs to be resized. Since this adjusts the position of every data piece, all entries in the GOT need to be updated.

Strings are a special case. There are no entries in the symbol table for strings and they are not in the data section with global variables and other data entries. Instead for security reasons, they are in a read-only data section and the GOT contains the appropriate offset entry without any associated entry in the symbol table. Instead, ATTUNE’s quilting procedure iterates through all strings in the old binary for the referenced string in the new binary.

**PLT and external reference quilting** In modern executables most libraries are dynamically linked, so instead of direct address entries, the ELF spec uses the procedure linkage table (PLT) to implement lazy loading. Only when the procedure is called does the dynamic linker locate the external symbol in question. In the event the new binary depends on an external symbol that isn’t in the original binary, as is the case when the updated functions depend on updated library versions, the PLT needs to be updated with the appropriate entry and the BPD metadata must include the appropriate information for the linking procedure.
6.4 Evaluation

We use a series of case studies to answer three main questions.

1. Can ATTUNE successfully resolve references across versions in real software?

2. How much new attack space does quilting introduce?

3. After the patched binary has been created can it be used to verify that both bugs have been patched and no side effects have been introduced?

We evaluated our prototype on a Dell OptiPlex 7040 with Intel core i7-6700 CPU at 3.4GHz with 32GB memory, running Ubuntu 18.04 64bit, using gcc/g++ version 7.4.0 and python 3.4.7. ATTUNE’s quilting procedure is built using CMake version 3.10.2 and Make version 4.1.

Resolving References We used ATTUNE’s BPD and quilting algorithm to construct quilted binaries across a variety of projects with a variety of different updates. Table 6.1 shows the bugs that were patched, a summary of the reason for the code change, the source code size of the change, the number of code section and data section resolutions ATTUNE made during the quilting procedure.

To test our quilting procedure with the updates from Table 6.1, we quilted the minimally changed patched version of the binary and the latest available buggy version. We also developed inputs that covered the patched code such that we could test that all references were resolved correctly. If there were any references that ATTUNE didn’t resolve correctly during the program run, the program would break. We found that the outputs of the quilted versions were consistent with the outputs of the updated versions. It should be noted that the number of resolutions ATTUNE requires is independent of the size of the patch but instead dependent upon how central the modified code is to the program’s control flow.

Mathematical Errors: Libpng In some applications mathematical errors have security implications, causing pointer errors or integer overflows. In this instance, an attacker could craft a malicious PNG image that triggers a bad calculation of \( \text{row\_factor} \) in Figure 6.5 [154]. This causes a divide-by-zero error and Denial-of-Service (DoS). After the developer writes the patch and builds
<table>
<thead>
<tr>
<th>Update</th>
<th>Patching Effort</th>
<th>LOC Changed</th>
<th>Code Resolutions</th>
<th>Data Resolutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl [30]</td>
<td>Changes how a string is parsed</td>
<td>16+, 16-</td>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>curl [31]</td>
<td>Changes function arguments and call.</td>
<td>9+, 9-</td>
<td>318</td>
<td>69</td>
</tr>
<tr>
<td>libpng [154]</td>
<td>Calculation modification for divide by 0 error</td>
<td>6+, 3-</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>wc [181]</td>
<td>Added new function and changed condition check</td>
<td>23+, 2-</td>
<td>298</td>
<td>109</td>
</tr>
<tr>
<td>yes [194]</td>
<td>Substantial changes in option parsing</td>
<td>40+, 141-</td>
<td>399</td>
<td>234</td>
</tr>
<tr>
<td>ls [111]</td>
<td>Added condition for change in option parsing</td>
<td>1+, 2-</td>
<td>387</td>
<td>380</td>
</tr>
<tr>
<td>mv [1]</td>
<td>Adding a conditional check before operation</td>
<td>6+, 0-</td>
<td>204</td>
<td>89</td>
</tr>
<tr>
<td>df [38]</td>
<td>Replacing open calls with stat calls</td>
<td>12+,8-</td>
<td>348</td>
<td>164</td>
</tr>
<tr>
<td>bs [157]</td>
<td>Changing a loop condition</td>
<td>2+, 1-</td>
<td>296</td>
<td>140</td>
</tr>
<tr>
<td>wget [162]</td>
<td>Adding conditional check for log</td>
<td>1-, 2+</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>redis [156]</td>
<td>Adding conditional check</td>
<td>1+, 1-</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

*Table 6.1: Patch-Testing Dataset*
the new binary, ATTUNE will automatically generate the patch metadata for the quilting procedure. Then

ATTUNE builds a quilted binary and updates any software running an old version of libpng. When the quilted binary was tested with a maliciously crafted image, where the row_factor was no longer 0, it correctly handled the malicious image the same way as the updated version.

String Parsing: libcurl String parsing is tricky since there are many corner cases. Figure 6.6 [30] adjusts Curl’s treatment of URLs that end in a single colon. In the buggy version, Curl incorrectly throws an error and never initiates a valid HTTP request. Figure 6.6 shows the patch, ATTUNE resolves all the references in the binary, and sends a valid request.

We test the quilted binary using a specially crafted input and the execution recreates the context that triggered the bug, and then jumps to the patched code upon entering the modified function.

New Function Refactoring: ATTUNE even supports changes that introduce substantial refactor-

```c
+/* Return non zero if a non breaking space. */
+ static int iswnbspace (wint_t wc) {
+ return ! posixly_correct && (wc == 0x00A0 ...
+ static int isnbspace (int c) {
+ return iswnbspace (btowc (c));
+}
+ wc (args) {
- if (iswspace (wide_char))
+ if (iswspace (wide_char) || iswnbspace(wide_char))
goto mb_word_separator;
...
- if (isspace (to_uchar (p[-1])))
+ if (isspace (to_uchar (p[-1])))
+ || iswspace (to_uchar (p[-1])))
+ goto word_separator;
}
...
```

Figure 6.7: wc New Function and Refactoring
ing across the entire code base. This includes adding new functions. The new function is treated as new code, and the functions that call the newly implemented function are replaced as well. The quilted binary is tested by providing input from a specially crafted file. The quilted binary acts the same as the updated version proving it successfully integrates the patch into the old binary.

**Increased Attack Surface** By quilting the binary with new code, we introduce a new attack vector that an attacker could potentially exploit. However, since we only add the bare minimum code necessary to create the quilted binary even if the update involves including new versions of libraries our code only quilts the updated functions and its dependencies. Table 6.2 shows the minute increased attack surface. The maximum was measured at 14% and the minimum at .003%. Egalito keeps the additional attack surface to a minimum. Since Egalito raises the binary to an intermediate representation and then regenerates the binary, there are no additional jump instructions, trampolines, or extra pieces from the patched version of the binary or its accompanying libraries. Using binary regeneration instead of traditional binary rewriting mechanisms, even large transformations introduce minimal overhead.

**Side Effect Verification** The main advantage of quilting is that it prevents updates from unintentionally introducing unwanted side effects. Developers can test for most functionality-related side effects, but side effects specific to the user environment like performance and scalability are difficult to diagnose with deploying the new version or at least a mock deployment. Developers can’t test for these types of problems.

To simulate this we conducted a case study using the Redis webserver. A version of redis [156] had a bug in which connecting via a monitor thread, and sending a specific message caused the server to crash. This is a critical-level vulnerability that would need to be patched immediately.

To simulate the patch, we first quilted the binary in question and ran the same patch quilting procedure. We then tested the patch’s functionality with the predesigned test scenario to make sure that it successfully integrated the patch the same way we did in the other studies. We then ran Redis’s included benchmark commonly used to test Redis’ performance on the original, quilted, and updated binary. We found no evidence that quilting introduces any performance overhead. For
<table>
<thead>
<tr>
<th>Update</th>
<th>Original Size (KB)</th>
<th>Quilt Increase (KB)</th>
<th>Pctg Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl [30]</td>
<td>663.4</td>
<td>.687</td>
<td>.104</td>
</tr>
<tr>
<td>curl [31]</td>
<td>663.5</td>
<td>9.6675</td>
<td>1.4569</td>
</tr>
<tr>
<td>libpng [154]</td>
<td>942.2</td>
<td>.237</td>
<td>.025</td>
</tr>
<tr>
<td>wc [181]</td>
<td>222.0</td>
<td>5.941</td>
<td>2.675</td>
</tr>
<tr>
<td>yes [194]</td>
<td>148.5</td>
<td>10.669</td>
<td>7.184</td>
</tr>
<tr>
<td>ls [111]</td>
<td>623.5</td>
<td>7.497</td>
<td>1.202</td>
</tr>
<tr>
<td>mv [1]</td>
<td>613.5</td>
<td>4.303</td>
<td>.701</td>
</tr>
<tr>
<td>df [38]</td>
<td>423.3</td>
<td>5.971</td>
<td>1.41</td>
</tr>
<tr>
<td>bs [157]</td>
<td>295.1</td>
<td>5.648</td>
<td>1.914</td>
</tr>
<tr>
<td>wget [162]</td>
<td>8740</td>
<td>.303</td>
<td>.003</td>
</tr>
<tr>
<td>redis [156]</td>
<td>1086</td>
<td>.138</td>
<td>.013</td>
</tr>
</tbody>
</table>

Table 6.2: Quilting Overhead

100k requests and 50k clients, the original ran 5.19s, quilted 5.14s and updated 5.12s. For 200k requests and 100k clients, we recorded 10.43s original, 10.44s quilted, and 10.42s updated.
6.5 Limitations

ATTUNE depends on the developers having good practice in their developer commits. If they include extraneous code not related to the designated bug-fix in the same commit, ATTUNE would automatically include this extra information in the diffs. In other words, in order for ATTUNE to be useful, developers need to make modular commits that distinguish bug-fix commits from other commits containing enhancements and other changes, and the developers run their build process with BPD adding only the bug-fix commit to the buggy release. In other words, in order for ATTUNE to be useful, it relies on developers making modular commits that contain only changes related to the bug fix or enhancement. Poor development practice has large ramifications for ATTUNE’s usefulness. ATTUNE also relies on having access to the symbols in the build process. Of course, these are stripped for deployment but should be available in the developer environment.

ATTUNE has some approach limitations so it cannot accommodate arbitrary code changes. The main approach limitation is in dealing with data structure modifications. ATTUNE makes no attempt to track data types, so quilting in a new data type with the current implementation would be impossible.

The current ATTUNE implementation does not support changes to macros. Macros are inlined to the code sections, so this information would not be available in the binary. Attempting to find the location of macros in a binary is impossible since this information is lost in the developer’s build process. Therefore Egalito can’t apply the appropriate transformations. Similarly in code that is highly optimized, as would be expected in production-level code, this could cause a problem with inlined functions. Egalito would have no way of finding the locations of the inlined functions. In principle, ATTUNE’s components could be extended to recover macro and inlining metadata, and leverage that metadata during quilting, resp., but corresponding changes to Egalito would also be needed.

ATTUNE inherently relies on Egalito’s parsing ability to extract the semantic meanings of program structure from raw binary information, and so ATTUNE inherits Egalito’s parsing limitations
(mostly requiring relocation information) but if another binary regeneration engine parsed the binary there would be no problem.

We asked an independent team of three students to collect a dataset to construct a set of known C/Linux bugs that had already been fixed, 2016–2019, and to write appropriate test cases. Although we encouraged them to include some security vulnerabilities, we placed no restrictions on what bugs they could look for so the collected bugs may not be representative of bugs found in the wild that users would deem critical to patch.
6.6 Conclusion and Future Work

We showed that we can use binary patch decomposition (BPD) in conjunction with binary quilting (ATTUNE’s quilting function) to combine updated and older versions of a binary, post compilation, to create a variant of the older binary that only includes the desired changes from the updated version. Initial investigations with our ATTUNE prototype indicate that this approach works with real binaries based on real updates and real build processes. We ran case studies across multiple programs with a variety of update types and sizes. We also found that our technique since it leverages binary regeneration introduces minimal extraneous code minimizing the extra attack vectors introduced in the quilting process. Our case studies which suggest that ATTUNE could accommodate production updates without introducing side effects, relative to the original binary, beyond the designated patch. Further, ATTUNE’s leveraging of Egalito’s binary regeneration avoids the performance overhead, relative to the original binary, common in other approaches to binary rewriting.

In the future, we hope to conduct a more quantitative and thorough evaluation of our techniques, with more types of updates, which we compare to the overhead introduced by other binary rewriting tools. We have only tested with the minimal patch size for making an update of limited scope, but there’s no reason our technique couldn’t accommodate composite version updates of a significantly larger scope: multiple independent patches and/or a series of bug fixes that work together. Given access to the proper dataset we would like to test this with legacy binaries that are especially relevant and hard to deal with. Lastly, with additional engineering effort, we expect ATTUNE could be augmented to handle data-related updates and changes.
Chapter 7: Discussion

7.1 Generating Unit Tests vs. Generating System Tests

ATTUNE is effective and efficient at achieving its goal, testing candidate patches for a post-deployment bug encountered in well-tested application software. However, it is not very useful for regression testing. Each generated ad hoc test is specific to a particular candidate patch. Say the developers add the ad hoc test for the accepted patch to their regression test suite. When the codebase changes again later, the test may become invalid if it depends on the code that has changed or been removed. Even when that part of the codebase hasn’t changed, the ad hoc test may not protect against regressions in other parts of the code that affect the program state at the time the patched code is reached during execution, because it will still test the patched code with the previously recorded program and system state. We suggest the community investigate how to build on the binary quilting infrastructure we developed for ATTUNE, together with buggy execution traces (again from rr), to automatically generate practical unit tests for regression testing that reflect the now-known user environment conditions that triggered the bug, as depicted in Figure 7.1.

One central issue underlies most of ATTUNE’s limitations: It recreates the buggy conditions by replaying to the point where the execution has diverged too much for emulation to continue (or the recorded trace ends). However, a unit test should depend only on the intermediate state directly before invoking the patched function. We need to automatically recreate the binary-level state before a patch, without the full application, taking into account that the patch may be exercised multiple times in multiple ways during application execution. The fixture setup must include any other program state and environment resources accessed, such as file descriptors, not just parameters. For a patched function that is only encountered once, this can be extracted directly from the trace. If the same function is exercised multiple times, we will analyze its new and old binaries and
its binary-level executions in the trace to produce a common fixture. We call this key idea test sculpting.

```c
... -1641,8 +1641,8 ... insert_regex (...);
if (error_message)
    error (1, 0, "%s", error_message);
- (*arg_ptr)++;  
our_pred->est_success_rate = ...
+ (*arg_ptr)++;  
return true; }
```

(a) Patch increments argument pointer after function call

```c
... -1641,8 +1641,8 ... insert_regex(...);
if (error_message)
    error(1, 0, "%s", error_message);
- (*arg_ptr)++;  
our_pred->est_success_rate = ...
+ (*arg_ptr)++;  
return true;
```

(b) Test case checks outputs of function call

**Figure 7.2:** Patch and test case to fix options parsing bug

The example in Figure 7.2, of a find bug (find.07b941b1) from the DbgBench dataset [174], increments the argument pointer before it is used. The fix is simple: The argument pointer should
be incremented after the function call to point to the next argument, as shown at the top. The bottom shows the test case. To be complete, the test needs to provide both parameters and state to the function under test and check both the function’s return value and external side-effects. Here this is the value of `arg_ptr` and returned from `insert_regex`.

This test demonstrates the three insights that make Automated Ad Hoc Test Generation possible: First, symbols function as interface opportunities between the test code and the code under test. If symbols are stripped, the binary can still be parsed to find function boundaries but we require the developer to identify which symbols in the binary have changed. Unlike our original Ad Hoc Test Generation, which monitors the execution to replay events at specific code locations, if we set up all inputs to the symbol under test correctly before execution we don’t need to monitor the execution once the function has been called. The test is no longer tied to a candidate patch since the rest of the application is not part of the test, and the test will still perform (during regression testing) after the function under test is updated. Second, the test doesn’t require the whole trace, but only the inputs relevant to the symbol under test. Since the trace contains all the inputs to the function, those pieces of the intermediate state can be extracted without help from the developer. Third, unit testing frameworks provide hooks that can be called at any point in the execution without altering state. This means binary rewriting can be extended to automatically add the desired checks.

Since the same or different user organizations might provide multiple traces that expose the same bug or exercise the same function(s), further investigation is needed to determine how to combine the information sources, inspired by the strategies discussed by Elbaum et al. [44] for carving unit tests from developers’ system tests. Further investigation is required to determine how to generate unit tests for code clones not covered by the buggy execution trace but identified by the developer as requiring essentially the same fix. We aim for unit tests to be independent of each other, so dependency-ordering should never cause flakiness [196].

Our preliminary design for a system that accomplishes this is depicted in Figure 7.3. Tests will be generated as assembly instructions, to directly incorporate details from the recorded execution. One way to do this would be to use existing decompilation tools to convert to more readable source
Figure 7.3: Preliminary Test Generation Design

code; One could use Ghidra [53] and DIRE [97]. However, all example tests in this discussion were manually coded in C. The decompiled code can be augmented with DIRECT for clarity, and developers may modify the test code, e.g., to address additional features added later – but manual changes and even recompilation of the baseline unit tests may not accurately portray the corner case from the buggy execution trace [17].

Such a system would offer distinct advantages for targeted maintenance scenarios. Since all nondeterminism during the original buggy execution is accounted for in the recorded execution trace, these tests will never be flaky [196]. Both setup and teardown for each test would be constructed automatically, condensing the replay engine emulation from the previous modified symbol. These tests would be robust since they won’t depend on any specific software version, they would be reusable since they would have built-in assertions, and they would be maintainable since they would run from regular unit testing frameworks.

Test cases need to accept patches that fix the bug and reject patches that do not, but some patches may fix how the bug manifests without fixing the underlying behavior. Figure 7.4 shows another find bug (find.183115d0), which deals with file descriptor leaks. This bug may not be obvious until reaching the file descriptor limit. The correct fix shown at the top closes each file descriptor after use. This bug and its patch present different challenges to those from Figure 7.2, since the bug
@@ -657,6 +657,12 @@ impl_pred_exec (const char *pathname,
     if (local) { close(execp->wd_for_exec->desc); }
 if (target != pathname) { assert (local); }
 }

(a) Correctly closes file descriptors after use

TEST_CASE("File descriptors don't reach max", "[find_files]") {
    char* test_path = "test_path";
    /* directory setup with 10 files */
    Int dirfd = open(dir);
    Pid_t Pid = getpid();
    Struct rlimit;
    Rlimit.rlim_max = 7;
    /* extract vals of stat_buf, pred_ptr,
       prefix, pfxlen from recording */
    set_rlimit(pid, RLIMIT_NOFILE, rlimit);
    res = impl_pred_exec(dirfd, test_path,
       stat_buf, pred_ptr, prefix, pfxlen)
    ASSERT(!res)
    /* remove files */
}

(b) Test case checking function no longer fails

Figure 7.4: Program reaches file descriptor limit
depends on the system state outside the program under test.

What makes this system possible is all this state is in the buggy trace from the user environment. rr’s verbose traces include file descriptor accesses (both local files and network streams) and environment variables like maximum allowed file descriptors. While a virtual machine or container could recreate the entire user environment state, the verbose trace is enough for the state this and most tests need.

7.1.1 Feasibility Studies:

We first checked whether our existing ATTUNE tool could generate ad hoc tests for all patches included in DbgBench [174]. Then we considered 1) Is the information in the recorded traces sufficient to automatically generate unit tests? 2) Do these tests accept the correct patches and reject the incorrect patches? (Recall that DbgBench includes both correct patches and “plausible” but incorrect patches.) From the recorded buggy execution traces, we identified all external inputs and extracted their values from the trace. We also extracted any events (like system calls) that occurred during the buggy executions. Then we manually coded corresponding unit tests, with assertions supplied by the DbgBench dataset. Additional research could investigate how to automate all of this.

<table>
<thead>
<tr>
<th>Project Name</th>
<th>grep</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Bugs</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>Ad Hoc Test Generation</td>
<td>84.6%</td>
<td>100%</td>
</tr>
<tr>
<td>Unit Test Generation</td>
<td>84.6%</td>
<td>92.8%</td>
</tr>
</tbody>
</table>

*Table 7.1: Test Generation Feasibility*

ATTUNE successfully constructed replay-based ad hoc tests for 84.6% of DbgBench’s grep bugs and 100% of its find bugs, see Table 7.1. For every grep bug where ATTUNE succeeded, we could manually construct a unit test. We could construct unit tests for every find bug except one (find.b445af98): This bug-fix makes an additional function call, not in the original code, that references an external state not included in the recorded execution trace. This was ok for ATTUNE since the emulated replay continued far enough for the developer to check, but not ok for automated
unit testing. The theorized system would need to recognize such cases, and inform the developer when new dependencies make it impossible to automatically generate complete tests. It could, however, generate test skeletons with placeholders for the developer to insert code. We used the assertions from DbgBench; every unit test correctly accepted every correct patch and correctly rejected every incorrect patch. We used the Catch2 testing framework [131] (future research would need plugins for each testing framework).

7.2 Adding Assertions for Side-Effect Detection

```c
@@ -1641,7 +1641,7 @@ insert_regex (...);
 if (error_message)
   error (1, 0, "%s", error_message);
-(*arg_ptr)++;
+(*arg_ptr)++;/*
 our_pred->est_success_rate = ... 
 return true;
```

(a) Plausible patch that passes developer test

```c
TEST_CASE("Parses arguments correctly", "[regex_parsing]") {  
 Char **argv = ["-regex", ".a", "-L"];
 ... 
 ASSERT(insert_regex(argv, arg_ptr, parser_table, entry, 
   regex_options)/*==TRUE*/);
 ASSERT(arg_ptr == 2); /*implied assertion*/
} 
```

(b) Assertion on implied output

Figure 7.5: Using buggy execution trace as pseudo-oracle

Unit tests usually have assertions (test oracles) to check the intended functionality of the unit under test. The DbgBench study showed 34% of submitted patches were faulty even though they passed the previously failing developer’s test [174]. Developers protect their codebases against faulty bug fixes using QA techniques that missed the bugs in the first place. Their post-deployment bug-fix code reviews may be more intense, but a recent study showed that maliciously crafted kernel patches

128
could appear to fix bugs and pass community review, while covertly introducing serious security vulnerabilities [59]. We want to extend unit test generation with automatically generated assertions that protect against side effects from faulty patches.

When developers write tests, they are usually designed to capture a specific behavior. Earlier sections discuss when developers write specific assertions designed to protect against the buggy behavior identified in the field. However, the DbgBench study showed around 34% of the patches submitted by the studied practitioners were faulty but still fixed the buggy behavior [174]. Developers may protect their codebases against faulty fixes through code review, which can cause problems. So our targeted maintenance test generation techniques need to protect against the side-effects introduced by faulty (or malicious) patches. One could combat this problem by augmenting unit tests with assertions on side effects, including everything that should change to eliminate the buggy behavior as well as everything that should remain the same.

One class of "plausible" fixes diverge from the buggy code in incorrect ways. Recall the patch and test in Figure 7.2. Figure 7.5a shows a patch submitted by another study subject for the same bug. Instead of moving the argument pointer increment from before to after reading the argument, the correct fix, the increment is commented out. Nevertheless, this patch passed the developer assertion shown in 7.2b.

Developer-written assertions may not mention every output, such as persistent changes to the
program and environment state. Without specifications or models, our proposed toolkit cannot know what was *supposed to* happen to this state. But it can assume any outputs not checked by the assertions developers add to generated unit tests were already correct in the buggy version, and use its trace as a pseudo-oracle. Of course, this won’t always be an accurate assumption, developers need to check the generated assertions. Genthat [115], which extracts unit tests from (non-buggy) execution traces of client code using the target software package, likewise treats recorded outputs as oracles. Here, `arg_ptr` is modified by the original function so it should be considered an output. The output value in the recorded trace is used to construct the implied assertion in Figure 7.5b.

Another class of "plausible" fixes impacts the program and/or environment state differently than the buggy execution trace, but both are wrong. Consider the patch and associated test shown earlier in Figure 7.4. To fix the same bug, a different DbgBench participant submitted the patch in Figure 7.7a. Instead of closing file descriptors after use, the correct fix, this patch waits until the file descriptor limit is reached and then closes an arbitrary open file descriptor. Here we should not rely on the execution trace, where all the file descriptors were (incorrectly) left open: If an assertion were generated to ensure all the file descriptors were left open, hopefully, the developer could easily mark it invalid. But we can automatically check tentative assertions and generate other assertions following an extensible set of heuristics – what is open should be closed, what is locked should be unlocked, etc. Various tools like Valgrind [133] already do this, so it’s not novel, but we should include it anyway at least as a sanity check on tentative assertions. The augmented test case shown in Figure 7.7b checks that the function no longer has any file descriptors allocated after completion.

One could address these problems with a tool for generating assertions about side effects. Figure 7.6 depicts how such a system would augment automatically generated unit tests with additional assertions, beyond those provided by developers, to check implied outputs involving either program or environment state.
initialise_wd_for_exec (struct exec_val *execp, int cwd_fd, const char *dir)
{
    execp->wd_for_exec = xmalloc (sizeof (*execp->wd_for_exec));
    execp->wd_for_exec->name = NULL;
+    if (execp->wd_for_exec->desc >= 0)
+        close(execp->wd_for_exec->desc);
}

(a) Incorrect patch leaks file descriptors

TEST_CASE("Close file descriptors", ":[find_files]") {
    /* extract values from recording and state setup */
    set_rlimit(pid, RLIMIT_NOFILE, rlimit);
    impl_pred_exec(dirfd, test_path,
        stat_buf, pred_ptr, prefix, pfxlen);
    /* complete assertion to
      check more than just the code didn’t crash */
    Int count = 0;
    For (int i=0; i<rlimit.rlim_max; i++) {
        Int open = fcntl(i, F_GETFD);
        if(open) count++
    }
    ASSERT(!count/*==0*/)
/* remove files */
}

(b) Assertion on environment state

Figure 7.7: More accurate test assertion

<table>
<thead>
<tr>
<th>Project Name</th>
<th>grep</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plausible fixes with noticeable side-effects</td>
<td>36</td>
<td>19</td>
</tr>
<tr>
<td>Plausible fixes incorrectly alter program state</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>Plausible fixes incorrectly alter environment state</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Plausible fixes rejected by side-effect assertions</td>
<td>91.6%</td>
<td>89.4%</td>
</tr>
</tbody>
</table>

Table 7.2: Assertion Generation Feasibility
7.2.1 Feasibility Studies:

DbgBench has many labeled examples of plausible but incorrect fixes, along with the problematic test cases that passed the plausible fixes. We asked 1) Are changes affecting either program or environment state responsible for any of the plausible but incorrect fixes? and 2) Can we generate accurate and useful assertions based on prior behavior and heuristics? As a preliminary study, we again turned to DbgBench as it has many labeled examples of the plausible fixes we are trying to protect against along with the faulty test cases. We wanted to determine 1) Are changes affecting either programmatic outputs or environmental outputs as described above responsible for these plausible but incorrect fixes? and 2) Can we add the additional assertions based on prior behavior and basic heuristics to fix them?

We manually went through all of the plausible fixes in the DbgBench dataset and analyzed them to see if they were deemed unacceptable due to programmatic side effects, environmental side effects, or some other reason. For each bug we manually extracted the function outputs and constructed the additional assertions required. Then we evaluated if those assertions identified the plausible fixes as incorrect, as shown in Table 7.2.

72% and 43.2% of plausible fixes for grep and find, resp., were not accepted by the DbgBench authors due to flaws visible without more test cases (with different inputs). We augmented each of these tests to check side effects. Our manual additions caught 91.6% and 89.4% of plausible yet incorrect fixes for find and grep, as shown in Table 7.2. Some plausible yet incorrect fixes that we missed are unlikely to be caught with test assertions, e.g., when the developer unnecessarily changes a third-party library. Other misses may become catchable with further work.
Epilogue

This thesis has presented a series of tools and techniques that enhance the traditional software maintenance procedure to accommodate the difficulties of maintaining software in practice, as well as a discussion of some future directions. While I am sure that readers will have many opinions about the efficacy and applicability of these techniques, I hope that this work encourages the community to rethink some of the traditional procedures and processes. For the reader, I appreciate your time and attention and I hope this work helped you in some way. Sincerely, Anthony
References


A. Fog, *Calling conventions for different C++ compilers and operating systems*. Technical University of Denmark, 2019.


Running `b2sum` with the `–check` option, and simply providing a string "BLAKE2". 


