Improving Security Tasks Using Compiler Provenance Information Recovered At the Binary-Level

Yufei Du
Georgia Institute of Technology
Atlanta, Georgia, USA

Omar Alrawi
Georgia Institute of Technology
Atlanta, Georgia, USA

Kevin Snow
Zeropoint Dynamics
Chapel Hill, North Carolina, USA

Manos Antonakakis
Georgia Institute of Technology
Atlanta, Georgia, USA

Fabian Monrose
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

The complex optimizations supported by modern compilers allow for compiler provenance recovery at many levels. For instance, it is possible to identify the compiler family and optimization level used when building a binary, as well as the individual compiler passes applied to functions within the binary. Yet, many downstream applications of compiler provenance remain unexplored. To bridge that gap, we train and evaluate a multi-label compiler provenance model on data collected from over 27,000 programs built using LLVM 14, and apply the model to a number of security-related tasks. Our approach considers 68 distinct compiler passes and achieves an average F-1 score of 84.4%. We first use the model to examine the magnitude of compiler-induced vulnerabilities, identifying 53 information leak bugs in 10 popular projects. We also show that several compiler optimization passes introduce a substantial amount of functional code reuse gadgets that negatively impact security. Beyond vulnerability detection, we evaluate other security applications, including using recovered provenance information to verify the correctness of Rich header data in Windows binaries (e.g., forensic analysis), as well as for binary decomposition tasks (e.g., third party library detection).

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; *Software reverse engineering*; • **Software and its engineering** → **Compilers**; *Software testing and debugging*; • **Theory of computation** → **Program analysis**.

KEYWORDS

Compiler-introduced security bugs; correctness-security gap

ACM Reference Format:
Yufei Du, Omar Alrawi, Kevin Snow, Manos Antonakakis, and Fabian Monrose. 2023. Improving Security Tasks Using Compiler Provenance Information Recovered At the Binary-Level. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623098

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623098

1 INTRODUCTION

Over the past decade, compilers have taken on a life of their own. Originally viewed as only a simple machine code generator, compilers are now also performing sanity checks, adding security features (*e.g.,* shadow call stacks), and applying a broad spectrum of optimizations. As compiler developers add more and more optimizations to each new release, programs enjoy free performance gains without sacrificing correctness or compatibility. LLVM, for example, includes new and improved optimizations through each version [29] and regularly updates target-specific optimizations to maximize performance for new hardware; similarly, the release notes of GCC [16, 17] show that each new release brings tens of improvements in optimizations such as vectorizer optimizations, inter-procedural optimizations, link-time optimizations, and target-specific optimizations. The advancements are happening so quickly that different compiler families, compiler versions, and optimization options can generate very different machine code from the same source code. But, unbeknownst to most programmers, that complexity might come at a cost to security.

In fact, recent work that explores the security impact of compiler optimizations [7, 13, 23, 48, 62] has shown that certain compiler optimizations can cause serious security issues. Specifically, D'Silva et al. [13] introduced the *correctness-security gap* and outlined how modern compilers perform optimizations that uphold the correctness of the program per programming language standards can nevertheless bring negative impacts to security, creating vulnerabilities such as information leakage and constant time violations.

One way to alert developers about these risks is to flag potential vulnerabilities that arise during the compilation process of their software or in the libraries they import. Indeed, an evaluation by Xu et al. [59] on bug reports and commit messages related to compiler-induced vulnerabilities found that although some developers are aware of security issues introduced by compiler optimizations, there are no effective solutions to help them detect or prevent such bugs. The lack of solutions may be because the recovery of provenance information is difficult because binaries do not include direct indicators of their compiler provenance.

Besides finding compiler-induced vulnerabilities, compiler configuration information is also valuable for other security applications. For one, malware analysts could use the compiler configuration information of malware to assist with authorship attribution [1, 8, 45], under the assumption that malware from the same author is more likely to share the same compiler and compiler options. Secondly, compiler configuration information could aid

critical binary decomposition tasks when constructing software bill of materials (SBOMs) [4]. The need for SBOM techniques [15, 63] is becoming increasingly important in light of widely publicized software supply chain attacks. Binary decomposition aims to dissect a binary file into smaller units, such as functions in the same object file or library. Because functions in the same object file typically share the same compiler configuration, coupled with the fact that it is common practice to build libraries separately from the main program, binary decomposition tools could use compiler configuration information of each function as a feature for grouping code.

Unfortunately, while compiler provenance recovery has been an active research area [20, 21, 46] with several applications — especially within the realm of binary code similarity [10, 31, 42, 52] — it was not until recently that our previous study [14] showed the possibility of recovering fine-grained compiler pass information at the function level using statistical learning techniques. That work shed light on a promising new direction for low-level compiler configuration recovery, although it was not without limitations. For one, the empirical evaluation was limited, and both the training and testing sets came from the same collection of programs.

In this paper, we go further by demonstrating the effectiveness of compiler provenance information in a wide variety of security tasks, including program analysis, bug detection, and forensic analysis. We begin by re-evaluating our previous approach on a significantly larger benchmark. Next, we extend the work by presenting a principled, large-scale evaluation on how compiler-induced vulnerabilities impact real-world projects. Specifically, we examine the magnitude of vulnerabilities introduced by the dead store elimination (DSE) optimization and also explore the extent to which code reuse gadgets are introduced by different optimization passes. We show how developers and analysts could use the recovered provenance information to focus on functions modified by risky optimizations. Additionally, we study a series of downstream tasks in program analysis that leverage compiler information recovery to improve security applications. To our knowledge, our study is the first to explore the security impact of compiler optimizations at the pass level and downstream tasks of function-level provenance information. Our specific contributions include:

- We demonstrate the generalizability of fine-grained pass-level recover via an evaluation on real-world programs built using LLVM 14 with separate training and testing data.
- We provide an in-depth look at the magnitude of compiler-induced vulnerabilities in 10 popular Linux C/C++ projects. As part of the study, we uncovered 53 instances of information leaks that can be attributed to DSE. For both C/C++ projects and Rust crates, we also evaluate the impact of compiler passes on gadget availability and observe that certain optimization passes, including the `Scalar Replacement of Aggregates (SROA)` and the `Control Flow Optimizer`, introduce a substantial amount of code reuse gadgets.
- We showcase applications that utilize compiler pass information for improving security. Specifically, we show how compiler artifacts enable the verification of the Rich header for Windows binaries and enable the assessment of the quality of binary decomposition algorithms. On the former, we show that our verification approach can identify binaries with forged Rich header information with high accuracy. On the latter, we show that our approach for fine-tuning binary decomposition algorithms generates robust and intuitive groupings compared to non-iterative approaches.

We organize the paper as follows. Section 2 introduces relevant background information on compiler optimization and compiler-induced vulnerabilities. Section 3 describes our methodology, including the corpora of programs we use and the information recovery technique we extend. Section 4 explores the magnitude of compiler-induced vulnerabilities, focusing on dead store elimination and code reuse gadgets. Section 5 presents applications related to forensics and binary decomposition, respectively. We discuss the implications of our work in Section 6, alongside limitations in Section 7. We conclude in Section 8.

2 BACKGROUND AND RELATED WORK

Modern compilers provide several command-line arguments that allow users to specify the optimization levels. For C/C++ compilers, the optimization levels are usually `-O0`, `-O1`, `-O2`, and `-O3`, where `-O0` means no optimizations and `-O3` means all optimizations. Internally, each optimization level maps to a list of optimization passes. The LLVM [30] compiler infrastructure, for example, utilizes front-ends (*e.g.,* Clang) to first convert the source code into the LLVM intermediate representation (LLVM IR). Then, it applies optimization passes at the LLVM IR and machine code levels.

It is important to keep in mind that at compile time, the compiler does not always apply all optimizations for the given optimization level. For example, in LLVM 14, there are at least 61 passes included at `-O1` optimization level, with an additional 13 passes at `-O2` and 2 more passes at `-O3`. Many are only applied if a specific code structure (*e.g.,* a loop) is present. More importantly, each optimization pass first determines if an optimization is beneficial. The compiler only applies the specific optimization if it can improve runtime performance and does not impact certain security features (*e.g.,* CFI and memory sanitizers). Therefore, knowing the optimization level of a binary is *not* enough to infer the set of passes applied. Even at the same optimization level, the compiler will subject each binary function to different sets of optimization passes.

Several studies [39, 52] have proposed techniques to identify artifacts like the compiler version (*e.g.,* GCC vs. Clang) and optimization level (*e.g.,* `O1` vs. `O3`) of stripped binaries. Recently, we presented a shallow learning technique [14] for identifying the application of individual compiler passes (*e.g.,* early CSE or peephole optimization) at a function level. While promising, that evaluation was conducted on a dataset with only four open-source projects, leaving questions about the broader applicability of the approach.

The Correctness-security Gap. D'Silva et al. [13] introduced the notion of the correctness-security gap where the compiler generates correct code per the language specification, but it introduces security issues. The authors categorized three types of compiler-induced security issues, namely, persistent state violation, undefined behavior, and side-channel attacks. A persistent state violation occurs when a memory object persists in the memory beyond its designed scope. A common optimization that causes this violation is the dead store elimination (DSE), which may remove instructions that

erase secret data, causing the secret data to persist in the memory even though the programmer explicitly erased it in the source code. The usage of undefined behavior could also lead to security issues. Another example is overflow detection using signed integer comparisons, where the programmer checks if a known positive integer is larger than a negative constant. Compiler optimizations could also make side-channel attacks possible. Cryptographers often apply constant time techniques for cryptographic operations to ensure the control flow and energy consumption stay the same regardless of the value of a secret. However, compilers may detect the semantics of such operations and apply optimizations that make code run faster, thereby enabling a side-channel attack [53].

Hohnka et al. [23] subsequently showed that the correctness-security gap exists in both the GCC and the Visual Studio compilers. However, like D'Silva et al. [13], Hohnka et al. [23] demonstrated the problem using specially crafted code snippets without testing real-world applications. Simon et al. [48] focused on studying the impact of certain optimizations on persistent state violations and side channels. To do so, they composed multiple variants of an example constant time function, compiled them with different versions of the Clang compiler, and demonstrated that newer versions of the compiler could optimize more variants of the constant time function, thus, being more likely to introduce side channels.

Brown et al. [7] studied the relationship between compiler optimization levels and the amount of code reuse gadgets in the compiled program. Their experiments show that at any optimization level higher than `O0`, the GCC compiler introduces more code reuse gadgets and would generate gadgets with undesired side effects. The authors go on to show that for certain optimization flags, the compiler introduces more special purpose gadgets (such as system call gadgets) — but they fell short in showing *which optimization passes* induce the observed behavior. In fact, they conclude that "avoiding negative security impacts on CRA gadget sets is not as simple as selecting a particular optimization level" [7].

More germane is the recent study by Xu et al. [59] that examined known compiler-induced security bugs by sifting through bug reports and commit logs. Their study highlights two common root causes for these bugs: *implicit* and *orthogonal* specification. Implicit specification occurs when the compiler makes implicit assumptions that differ from the developer's intention; for example, the compiler may interpret undefined behaviors differently from the developer's will. Orthogonal specification occurs when the language standard lacks certain security-related concepts such as constant time guarantees or memory lifetime; constant time violations and information leaks due to dead store elimination are prominent examples of this root cause. That work is concurrent with ours and, taken together, sheds light on the prevalence of the correctness-security gap. Furthermore, the authors studied methods to mitigate compiler-induced vulnerabilities and concluded that the available methods all have major drawbacks. Lastly, their study shows that while developers sometimes follow code practices for writing code that thwart optimizations, compilers are also becoming increasingly intelligent and often proceed to apply the optimizations anyway — further motivating the need for the approach we propose to detect implicit violations.

3 SYSTEM DESIGN

Unlike prior works, our approach creates a robust and more generalizable model that can be directly used in real-world applications, including forensic analysis, vulnerability analysis, and library identification. To do so, we train an architecture specific machine learning model to identify compiler optimization passes applied to binary files. Figure 1 provides an overview of our approach. First, we use an instrumented compiler (❶) to generate the ground truth of compiler pass information applied to each function during the compilation process. Then, we disassemble the compiled binary and extract features (❷) on the corpus of functions. Afterwards, we use the compiler pass ground truth information to perform feature selection and generate a labelled dataset for training (❸). Finally, we use statistical learning techniques to build models for each compiler pass (❹). We describe each in turn.

3.1 Approach

Compiler Instrumentation. We modified the LLVM compiler to obtain a list of compiler passes that alter a function during the compilation process. Our modifications to the compiler do not interfere with code generation in any way, and a program built by the modified compiler is identical to that built using the same version of the original LLVM compiler with the same flags. In order to accurately identify the list of passes that optimize a code segment, we incorporated code to log the passes that modify functions of a program during the compilation process. Note that throughout our workflow of compiler pass information recovery, we only use the information generated by the modified compiler as ground truth for training and validation. We also log the instructions that the Dead Store Elimination pass eliminated and recorded the corresponding reasons for their removal. For ground truth, we generate a pass log file and the target binary for each program we compile. The pass log file records pass types executed at all levels, including the *ModulePass*, *CallGraphSCCPass*, *FunctionPass*, *LoopPass*, and *MachineFunctionPass* types. Each pass type modifies the code at various levels. For example, *ModulePass* and *CallGraphSCCPass* are inter-procedural optimizations for source files and call graphs. *FunctionPass* and *LoopPass*, on the other hand, are intra-procedural optimizations for individual functions and loops within functions, respectively. The log file contains the list of compiler passes that modify the code, and for intra-procedural passes, the log file includes the function name that the pass modifies.

Feature Type	Example
Opcode	`mul`
Instruction	`mov #MEM# %rip %r15`
Register	`r11w`
n-gram of opcodes	`cmp | cmove`
n-gram of instructions	`jmp #TARGET# | lea #MEM# %rip %rdi`
First instruction	`START test %edx %edx`
Last instruction	`END cmp #MEM# %rbx %edx`

Table 1: Feature types used for optimization pass recovery

Feature Extraction. Following the compilation process, the next phase entails disassembling the binary objects and extracting a set of features. We begin by disassembling the code sections of the

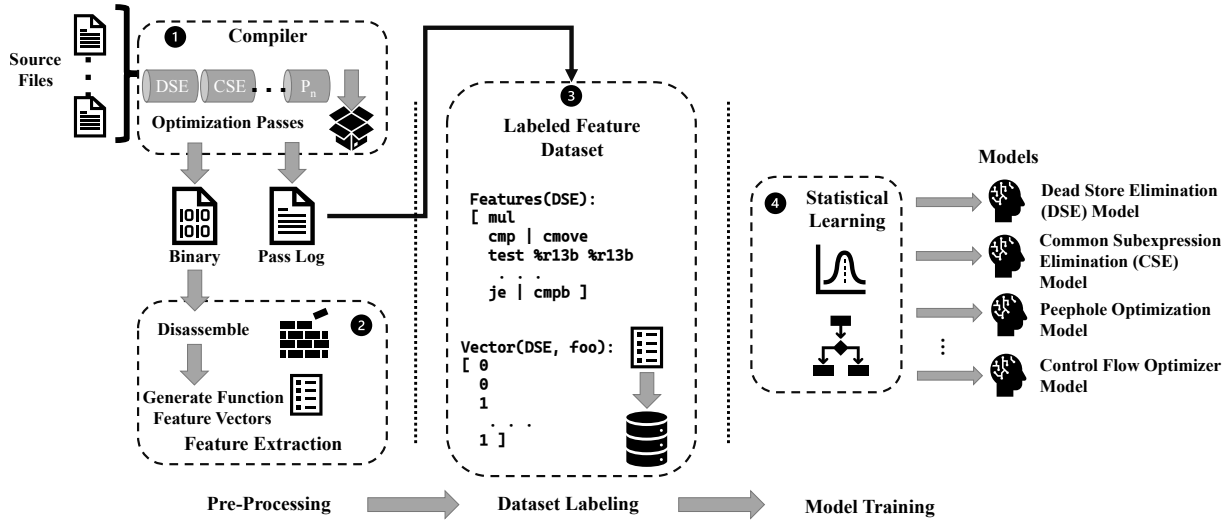

Figure 1: An overview of the model building and training.

binary file and transforming the disassembled code into a list of instructions. We then extract features using this list. Table 1 shows the type of features we use. Each feature captures the presence of a specific code pattern. A feature vector is the collection of features in a function, where each value is either True if the function includes the pattern or False otherwise. We use an n-gram encoding, with n ranging from one to three, to represent features based on instructions and opcodes. For instructions, we normalize memory addresses to `#MEM#`, call targets to `#TARGET#`, and immediate values to `#IMM#`. Each register is a feature. We also include the first and last instructions within a function as features because certain passes could change a function's prologue and epilogue. The feature vector values are binary (as opposed to being represented as a bag of words or using Word2Vec [34]) to deal with the fact that compiler optimizations can change the ordering of instructions and the specific opcodes and registers used.

Feature Dataset Labeling. We combine the features of each function with the compiler pass labels to generate the feature dataset we use for training. We associate the list of compiler passes applied to each function to generate the training labels. Utilizing the pass log collected during the compiler phase, we isolate function-level optimization passes (`FunctionPass`, `LoopPass`, and `MachineFunctionPass`) and discard inter-procedural ones. Some functions may have multiple labels resulting from different optimization passes. We incorporate these labels in the feature dataset, where each entry contains the project name, binary filename, function name, optimization level of the binary, the features, and the list of compiler passes applied to the function.

Statistical Learning. We use a multi-label (i.e., one function could be mapped to multiple passes) classifier that contains a collection of models. We train a separate model for each compiler pass using an approach based on a tree-based learning algorithm. In our case, the models are binary classifiers, predicting whether or not the compiler applied a specific optimization pass to a given function. We apply feature selection to limit the total amount of features we use for each model to improve performance. For feature types with an overwhelming amount of features (such as 3-gram of instructions), we select the m=1,000 most frequent features corresponding to an optimization pass. Since we train multiple models, each dedicated to a specific optimization pass, the importance of features varies among models. For instance, in the case of the `Post-RA Pseudo Instruction Expansion` pass, the most important feature is `START movslq %ecx %rax`, indicating that the function commences with this sequence. Conversely, the most important feature for the `Combine Redundant Instructions` pass is `mov %10 #MEM# %rsp | mov %rax #MEM# %rsp` representing a 2-gram of instructions. Appendix B lists the top three features for the 14 optimization passes listed in Table 3.

To ensure we use balanced training data, we randomly select a set of functions from the training set consisting of an equal number of optimized and non-optimized functions. For instance, for popular passes applied to numerous functions, we randomly select 15,000 functions with an optimization pass and another 15,000 without an optimization pass.

3.2 Model Evaluation

To conduct our evaluations, we curated a diverse set of binaries from projects listed in Table 2: popular Linux C/C++ programs and libraries, popular Windows C/C++ programs and libraries, and popular Rust crates. For all binary corpora, we built the programs and libraries as X86-64 binaries. For Linux C/C++ programs and libraries, we collected our binaries from two sources. First, we collected the programs included in the OSS-Fuzz project [19] (Corpus I). Second, we use the programs found in the firmware of the ASUS RT-AC88U router (Corpus II). We retrieved the list of open-source programs by analyzing the firmware binary of the ASUS RT-AC88U router with `binwalk` [43]. We then downloaded the source code of the latest version of these programs and built them to X86-64 binaries. We only include projects that we could successfully build using LLVM. Finally, we use the dataset from our previous work

[14] (Corpus III), which has been pre-processed and is in the format of a CSV file. For all the Linux C/C++ programs whose source code is available (Corpora I and II), we compiled these projects using optimization levels `O0`, `O1`, `O2` and `O3`. This resulted in a total of 12,576 binaries for Corpus I and 5,599 for Corpus II across all builds.

Corpus	Data Source	Language	Platform	Project Count
I	OSS-Fuzz	C/C++	Linux	36
II	X86-64 Router Programs	C/C++	Linux	33
III	Du et al. [14]	C/C++	Linux	4
IV	conan.io	C/C++	Windows	235
V	crates.io	Rust	Linux	200

Table 2: A summary of our corpuses of programs.

For Windows programs and libraries, we retrieved all open-source repositories for programs that support Conan [25], a C/C++ package manager (Corpus IV). Similar to the Linux binaries, we only include projects that we could build using the Microsoft Visual Studio compiler. For all programs in Corpus IV, we compiled them using three different versions of the Microsoft compiler and two optimization levels, namely `Debug` and `RelWithDebInfo` (Release with debug symbols). In total, we built 9,429 Windows binaries. Lastly, we pulled the top 200 Rust crates from `crates.io` as of November 2022 (Corpus V).

Experimental Setup. We implemented our pipeline using several tools. As noted earlier, we modified the `LLVM` compiler to generate the ground truth for the optimization passes. We use the same base version of LLVM (version 14 commit ID `c59ebe4`) as previously [14] to ensure that our compilation environment matches that of Corpus III. We used `objdump` to disassemble Linux binaries and generate the vector of instructions for each function.

We experimented with several learning models (see Appendix A) and ultimately choose to use LightGBM [27] as our multi-label classifier because it attained the best overall performance in compiler pass classification. The feature extraction tool and our classifier are written in Python. We use helper functions from Scikit-learn [38] to compute metrics. We train our multi-label model on a desktop machine with Core i9-12900KF processor and 64GB of RAM. For the evaluation of our classifier, we use Corpus I and Corpus III for training, while Corpus II serves as the testing set. This separation ensures that the model's performance is accurately assessed on unseen data. In total, the training set and the testing set include 140,938 and 65,605 unique functions, respectively.

3.2.1 Findings. Our evaluation uses two metrics: recall and the F-1 score. The recall measures the proportion of true positive predictions out of all actual positive instances, including true positives and false negatives. A high recall indicates few false negatives compared to true positives, helping us minimize false negatives. The F-1 score is the harmonic mean of precision and recall, combining both metrics into a single number. This F-1 score is instrumental when the costs of false positives and false negatives differ. In our design goals, false negatives are more impactful than false positives. For example, our classifier can be used to alert developers to instances where the compiler applies potentially harmful optimization passes to their code. A false negative means that our classifier missed a potentially harmful optimization pass, which could make the compiled program less secure. In contrast, a false positive occurs when our classifier incorrectly predicts a harmful optimization pass that does not impact the compiled program. Therefore, the recall score is more impactful than precision in the context of privacy-sensitive applications, as developers can manually verify and exclude false positive cases.

Optimization Pass	Training Samples	Testing Samples	[14] Recall (%)	New Recall (%)	New F-1 (%)
Dead Store Elimination	6735	994	85.8	71.7	59.9
Aggressive Dead Code Elimination	2322	500	83.5	77.6	67.8
Bit-Tracking Dead Code Elimination	8611	966	87.5	77.2	67.3
Remove Dead Machine Instructions	30000	1005	88.6	85.9	79.5
Early Machine Loop Invariant Code Motion	30000	1014	93.3	94.2	91.5
Machine Loop Invariant Code Motion	1755	426	89.0	86.3	80.1
Loop Invariant Code Motion	30000	995	90.6	91.3	87.2
Machine Common Subexpression Elimination	30000	999	88.1	89.1	84.1
Early CSE	30000	1018	92.2	91.9	88.1
Early CSE w/ MemorySSA	30000	962	88.6	84.9	78.0
Loop Strength Reduction	30000	1057	95.4	94.9	92.5
Peephole Optimizations	30000	993	98.0	95.6	93.6
SROA	30000	1028	-	99.1	98.7
Control Flow Optimizer	30000	984	-	98.3	97.6

Table 3: Compiler pass prediction. Orig. recall denotes the values reported in [14]. The '-' symbol means unreported. The two instances where recall differs significantly between the evaluations are highlighted in dark grey.

We evaluate our model on 68 different compiler passes. Table 3 presents the classification performance for a subset of the compiler passes. We choose this subset to compare with prior work [14] and to highlight the passes that introduce functional code reuse gadgets in compiled programs (see Section 4.2). Our technique achieved an average F-1 score of 84.4% across the 68 compiler passes. The use of distinct training and testing programs is likely the reason for the degradation in performance. The overall results show that the average recall is 89.1%. Most of the passes with large set of training samples achieve high F-1 scores. In Section 6, we discuss ways to enhance the classifier with additional contextual information.

Looking at the most significant features (see Appendix B) for the passes in Table 3, it is not difficult to see why the model performed well on some of the passes. For example, for the `Machine Common Subexpression Elimination` pass, the most significant feature is a single opcode, `punpckhqdq` (unpack and interleave high-order quadwords). As a `MachineFunctionPass`, this optimization pass directly modifies the X86-64 machine code to generate efficient code using this rare opcode. We surmise that this pass is one of the few, if not the only, pass that utilises this distinguishing opcode. Similarly, all top 3 features for the `Early Machine Loop Invariant Code Motion` pass are the first instruction of a function, with the one of the features including another rare opcode, `vcvtsd2ss`.

Lastly, we note that although our classifier does not perform as well as we would have liked on the DSE and Dead Code related

passes — due, in part, to the limited number of samples we have — we nevertheless explore if we can use the DSE model to flag potential problems in binaries from open source projects. Our motives for doing so stems from the fact that although the security community has raised the alarm about DSE [7, 13, 48, 62], there have been limited evaluations on real world data. We conduct such a study in the following section.

4 ON THE PREVALENCE OF COMPILER-INDUCED VULNERABILITIES

Since the detection of risky optimizations is a natural downstream application of compiler pass information recovery, we set out to empirically assess the scope of the secutity-correctness gap on real-world projects. To narrow the purview of that study, we focused on two scenarios that easily lend themselves to abuse: (*i*) when the dead store elimination optimization pass lets sensitive data persists in memory against the developer's will, and (*ii*) when optimization passes increase the amount of code reuse gadgets made available to an adversary. Our multi-label classifier puts us in a unique position to study these cases.

4.1 DSE Still Considered Very Harmful

For scenario (i), we use all 4,555 Linux binaries that were built at `-O3` optimization level in Corpora I and II. We pay attention to `memset` calls that are removed by the optimization before the end of the data structure's scope. Because LLVM converts known operations that perform sequential memory writes to `memset` calls before applying this optimization, focusing on `memset` is sufficient for our purposes.

As the `Dead Store Elimination` optimization pass removes some instructions that are irrelevant to our study, we apply filtering to the results to keep only `memset` instructions that write a constant value to a variable at the end of the variable's scope. We utilize two filters: first, since our instrumentation to the LLVM compiler tracks the reason why the pass removes an instruction, we ignore instruction removals that do not match our interest (*e.g.,* unnecessary variable initialization before it is overwritten by new values); second, we automatically inspect the source code corresponding to the removed instructions and check if the variable is not used again after the removed `memset` instruction.

4.1.1 Results. Using our approach to flag potential issues that arise during compilation, we discovered 53 removed dead stores across 10 projects that have security implications. Table 4 summarizes the results. Many affected projects are popular programs found on a wide range of devices such as `busybox`, a common Unix utility program found on most embedded-Linux devices, and `wpa_supplicant`, a Linux utility that allows the system to connect to WPA-secured wireless access points found on many distributions. In what follows, we discuss some of our findings.

Example I: Local Stack Data in `wpa_supplicant`. This utility handles the authentication for encrypted wireless networks and processes the access point key. Listing 1 shows part of the `wpa_supplicant_set_wpa_none_key` that processes the key. This function determines the type of the group cipher for the current wireless access point, writes the key to a temporary local variable, `key`, and then calls another function, `wpa_drv_set_key`, to store the key appropriately. Before this function returns, `wpa_supplicant_set_wpa_none_key` calls `os_memset` (a macro that redirects to `memset`) to erase the local variable `key` from the stack. However, at the `O3` optimization level, the Dead Store Elimination pass determines that this `memset` is unnecessary because `key` is not used anywhere after it is set to 0, causing sensitive data to persist on the stack. If an information leak happens between the time when this function returns and when the stack data is overwritten, the key may be exposed.

Project	Version/ Commit	Domain	Total Removed `memset`	Local Stack Data	Heap Data
Linux-PAM	1.5.2	Utility	18	9	9
Kamailio	961f276	SIP Server	11	11	0
Busybox	1.35.0	Utility	6	4	2
WPA_supplicant	2.10	Utility	5	5	0
Proftpd	4520af4	FTP Server	4	1	3
Netatalk	3.1.11	AFP Server	3	3	0
Varnish-cache	99607ac	HTTP Proxy	3	3	0
Tmux	85ef735	Terminal	1	0	1
e2fsprogs	96185e9	Utility	1	0	1
Lighttpd1.4	1.4.64	HTTP Server	1	1	0
Total			53	37	16

Table 4: Statistics of the removed dead stores by Project.

```
int wpa_supplicant_set_wpa_none_key(
    struct wpa_supplicant *wpa_s,
    struct wpa_ssid *ssid)
{
  u8 key[32];
  ...
  switch (wpa_s->group_cipher) {
  case WPA_CIPHER_CCMP:
    os_memcpy(key, ssid->psk, 16);
    ...
  case WPA_CIPHER_GCMP:
    os_memcpy(key, ssid->psk, 16);
    ...
  ...
  }
  ret = wpa_drv_set_key(wpa_s, alg, NULL, 0,
    1, seq, 6, key, keylen,
    KEY_FLAG_GROUP_RX_TX_DEFAULT);
  os_memset(key, 0, sizeof(key));
  return ret;
}
```

Listing 1: Code snippet from the `wpa_supplicant` utility

Including this example, we found a total of 37 instances where instructions that erase stack data were removed by the compiler's Dead Store Elimination pass. We manually confirmed that several of these vulnerabilities could be exploited.

Example II: Heap Data in `Linux-PAM`. Linux-PAM (Pluggable Authentication Modules) is a framework for user authentication in Linux. As the name suggests, Linux-PAM requires access to secret data including user passwords. Unfortunately, in multiple locations within Linux-PAM, the developers rely on `memset` to erase secret

data for both stack and heap memory. Here, we show an example of the removed `memset` that causes secret data to persist on the heap.

```
PAMH_ARG_DECL(int verify_pwd_hash,
  const char *p, char *hash,
  unsigned int nullok)
{
  ...
  struct crypt_data *cdata;
  cdata = malloc(sizeof(*cdata));
  if (cdata != NULL) {
    cdata->initialized = 0;
    pp = x_strdup(crypt_r(p, hash, cdata));
    memset(cdata, '\0', sizeof(*cdata));
    free(cdata);
  }
  ...
}
```

Listing 2: Code snippet from the `Linux-PAM` utility

Listing 2 shows a segment of the function `verify_pwd_hash`, one of the functions that Linux-PAM uses to verify a password. This function allocates a heap object, `cdata`, as a buffer for the `crypt_r` library function to hash the password `p`. After `crypt_r` returns, the function uses `memset` to erase the buffer `cdata` by filling the memory with `0` before freeing the buffer. However, since `cdata` is not used before being freed, the compiler determines that the `memset`, which is the last write to the buffer, is not needed and removes it. As a result, this buffer used by `crypt_r` would persist in the heap memory after `verify_pwd_hash` returns until the same heap memory is allocated and overwritten again. A heap overread exploit could leak the secret data stored in this buffer.

Concurrent to our study, Linux-PAM developers independently discovered and patched all but one of the removed dead stores we identified with our approach. For the outstanding undiscovered unsafe `memset` call in `sha1_process`, the developers released a new patch [9] based on our findings.

Including this example, we found a total of 16 instances where instructions that clear heap data were discarded by the compiler.

DISCUSSION. The fact that even popular security-sensitive programs such as Linux-PAM still includes code that could cause information leaks when built at high optimization levels underscore the severity of the threat posed by compiler-induced vulnerabilities and motivates the need for the technique we propose. Once a DSE-induced information leak is confirmed using our approach, developers can follow the recommendations of Yang et al. [62], such as using secure functions (*e.g.,* `memset_s`, `explicit_bzero`, and `SecureZeroMemory`) to fix the identified problems. Developers could also use the `volatile` attribute for memory operations in order to mitigate the threat from a compiler-induced vulnerability.

4.2 There Be Gadgets

Brown et al. [7] observed that compiler optimizations can introduce code that increases both the quantity and quality of code reuse gadget sets [44]. Their findings appear to show that this problem is ubiquitous across many compilers and optimizations levels, but their quantitative analysis was insufficient to draw conclusions about which specific behaviors caused negative security impacts. To dig deeper into the underlying causes, we set off to measure which passes played a pivotal role in introducing types of expressive gadgets that allow for critical actions, including register operations, memory operations, call gadgets, and stack pivots. We do so for both C/C++ and Rust programs. In the case of Rust, we note that while code reuse attacks such as return-oriented programming occur on languages that do not enforce type safety, previous studies [3, 58] have shown that memory safety bugs continue to exist in projects that utilize `unsafe`[37] Rust, making code reuse attacks theoretically possible there as well.

Experimental Setup. For scenario (ii), we use Corpora I, II and V. Since our goal is to find the optimization passes responsible for the issue, we need fine-grained control on running optimization passes during compilation such that we can evaluate the compiled binary with and without an optimization pass. Therefore, for C/C++ programs, we further modified the LLVM Clang compiler to generate a binary after each compiler pass runs such that we can measure the amount of gadgets before and after applying an optimization. Rust programs, however, cannot utilize the same feature in the compiler. Therefore, for Rust crates, we simply measure the difference in the amount of gadgets between the `Debug` build and the `Release` build.

We use the open-source gadget scanning tool Ropper [47] to find gadgets. Ropper supports grouping the gadgets into different types such as memory writes, memory loads, register loads and stack pivots. We utilize this feature to categorize the gadgets. For stack pivots, Ropper also includes gadgets that are not exploitable (*e.g.,* gadgets with only a return instruction with offset), so we exclude these gadgets in our analysis.

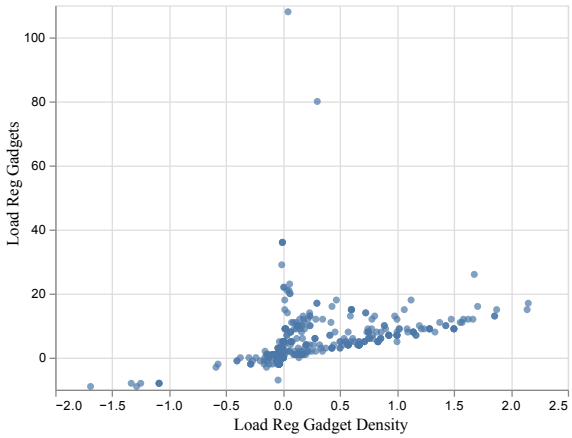

Figure 2: Changes in the number of gadgets that load a register after the SROA pass. The X-axis represents the density of the gadgets (i.e., amount of pivots divided by the code size in pages). The Y-axis shows the amount of gadgets. Each dot is a binary in Corpus I.

4.2.1 Results for C/C++. Overall, we confirm that the increase in gadgets is pervasive. For the Scalar Replacement of Aggregrates (SROA) function pass, in particular, we find that it frequently introduces gadgets that load stack data into registers (e.g., `pop r14;`

`ret;`). For example, Figure 2 depicts the changes in the amount of register loading gadgets and the density of gadgets for binaries in Corpus I. We show the gadget density in addition to the amount of register because advanced return-oriented programming attacks such as JIT-ROP [49] succeed faster in situations with high gadget density. There, the pass introduced 5.04 register loading gadgets on average with a median of 4 gadgets, and for binaries in Corpus II, this pass introduced an average of 6.98 gadgets of this type with a median of 5 gadgets. We observed that in the compiler pipeline, this pass always runs multiple times and introduces most of the new gadgets in the first iteration. We suspect that the compiler keeps running a specific set of function passes until the code converges or until a limit is reached.

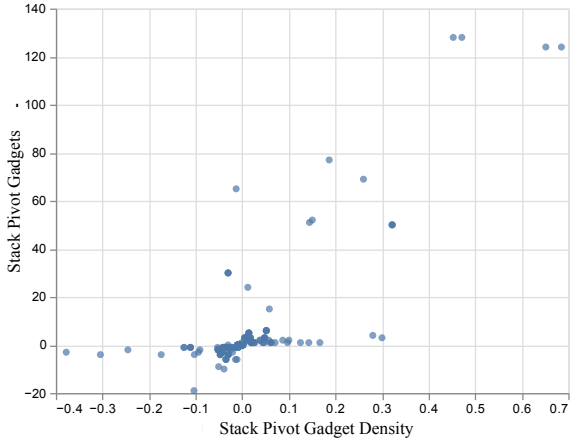

Figure 3: The impact of the SROA pass on the amount of stack pivot gadgets in binaries in Corpus I.

We also found function passes that introduced additional gadgets that manipulate register values. These results are provided in Appendix 9. More interesting is the fact that several passes show an impact on the amount of stack pivot gadgets. Stack pivot gadgets allows an attacker to change or "pivot" the stack pointer to an attacker-controlled location in memory. This gives attackers far more flexibility and control over the exploitation process, enabling them to bypass certain security measures (notably, address space layout randomization (ASLR) and stack canaries) to craft more effective ROP chains. For Corpus I, the first instance of `SROA` function pass introduced the most stack pivot gadgets among all runs of function passes. On average, this run of `SROA` added 2.53 stack pivot gadgets to each binary. Figure 3 shows the impact of SROA on the amount of stack pivot gadgets and the density of the gadgets. For Corpus II, three passes introduced a measurable amount of stack pivot gadgets. `Simplify the CFG`, a function pass that runs early in the compiler pipeline, added 2.33 stack pivot gadgets on average the first time it ran and introduced another 2.28 stack pivot gadgets on average later in the pipeline when it ran again. `EarlyCSE`, another function pass early in the pipeline, added 2.89 stack pivot gadgets on average. Other than function passes, one strongly connected component pass, the `Function Integration/Inlining` pass, also introduced stack pivot gadgets. On average, this pass added 3.72 stack pivot gadgets to each binary.

4.2.2 Results for Rust. We built projects from Corpus V using 4 recent versions of the Rust compiler released between 2021 and 2022, ranging from version 1.52 to 1.62. Figure 4 shows the average differences in the amount of gadgets for each gadget type. Surprisingly, instead of introducing more gadgets, the release optimization level causes the binary to contain *less* gadgets than the debug optimization level for most gadget types. The pattern is consistent across all five versions we tested. Among the ten gadget types, only three types, `increment register`, `subtract register` and `clear register`, see a measurable increase in the amount of gadgets when compiled with the release optimization level, but these gadget types are generally not as powerful [24] as the others.

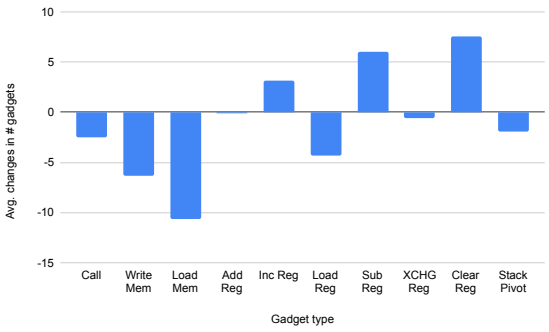

Figure 4: Average difference (between release and debug builds) for each gadget type among the top 200 Rust crates built using 4 versions of the Rust compiler.

To try to find a reason for the decrease of functional gadgets, we examined the list of passes that modified the code during compilation for these Rust crates at the release optimization level when built with the most recent version of the Rust compiler, 1.62. We found that the same optimization passes that introduce significantly more gadgets for C/C++ programs still execute for Rust crates. For example, the `SROA` pass that introduces more "load register" gadgets and stack pivot gadgets for C/C++ programs is also frequently applied to Rust functions. In the 200 Rust crates, the `SROA` pass was applied 799,987 times. Similarly, `Control Flow Optimizer`, the function pass that introduces more "increment register" gadgets for C/C++ programs, was applied 61,517 times in the Rust crates. Hence, even though the passes that introduce the most gadgets in C/C++ programs are still applied in Rust, they do not have the same impact on gadget availability when optimizing Rust programs. Unfortunately, given the limited documentation on the internal workings of the Rust compiler and the Mid-level IR (MIR) optimizations it applies, we cannot provide a definitive answer for why this behavior exists. That said, we suspect that the decrease in the amount of functional gadgets is likely a side effect of security instrumentation added by the Rust compiler, including type checking on the High-level IR (HIR), borrow checking on MIR, and unsafety checking on the recently introduced Typed High-level IR (THIR) [32, 51], before invoking the LLVM compiler. Given the dearth of information regarding these enhancements, the exact reasoning for the observed behavior remains an open problem.

DISCUSSION. Our study shows that for C/C++ programs, several optimization passes introduce functional gadget sets that are useful to an attacker. Unlike the case of information leak issues that could be mitigated by using special code patterns, an increase in code reuse gadgets is more challenging to address: the security impacts are difficult to quantify [7], and there does not appear to be a clear workaround for developers. One unsatisfying solution is to write everything in meticulously crafted assembly code. An equally unappealing option is to turn off all optimizations.

A third option is to use pass-level information to identify the code that was optimized and then use a binary recompiler [54, 55, 57] to patch the code. One option for doing just that is to use an x86_64 recompiler like Egalito [57]. Like other binary recompilers, Egalito was built with the goal of helping security-conscious developers trade security for performance in order to thwart specific threats (*e.g.,* advanced code reuse attack strategies [5, 49, 65]). To date, these binary recompilers have been used to implement JIT-shuffling protections and software-only control flow enforcement [57], defenses against speculative execution attacks, and even as a last ditch effort to disable Clang's tail call elimination optimization [7]. We leave the exercise of implementing plugins that rewrite risky code blocks identified by our approach as future work.

The situation is somewhat better with Rust, where the amount of expressive functional gadgets is low across compiler versions. But, Rust comes with its own set of challenges, especially at the foreign function interface (FFI) boundary where `unsafe` Rust is virtually unavoidable [2, 28, 58]. Moreover, for large applications, the rewrite or translation to memory safe languages must be done with care to avoid other vulnerabilities [33, 36].

We acknowledge that while it is important to consider the impacts of different optimizations on gadget availability, the existence of expressive gadget sets does not necessarily make a binary less secure.

5 REAL WORLD SECURITY APPLICATIONS

To showcase how one can leverage the recovered compiler provenance information, we now present two downstream applications. The first application flags discrepancies between the information contained in the *Rich header* of Windows binaries and the actual information we infer regarding the compilers used to build the binary under scrutiny. The second application highlights how low-level compiler pass information can be used to dissect a binary into smaller components comprising of the third party libraries it uses. We discuss each in turn.

5.1 Detecting Forged Rich Headers

Windows binaries built with Microsoft Visual Studio (VS) compiler include an undocumented section of the metadata, the Rich header, that includes the list of the ID of compilers, linkers, and other tools used when building binaries. Once the Rich header became more well known, malware analysts started using the compiler information within the Rich header as a supporting indicator in malware attribution [40, 56]. As expected, malware authors soon started to manipulate the Rich header to throw off threat hunters. Motivated by an incident [18] where the authors of the OlympicDestroyer malware fabricated the Rich header to impersonate the Lazarus group, we built a tool based on compiler provenance that detects binaries with tampered compiler ID information. The report indicates that the authors deliberately copied the Rich header from another malware family to mislead attribution. By comparing OlympicDestroyer to the mimicked malware, the analyst found an exact match between the Rich headers but a discrepancy in the code structure. The analysts manually reverse-engineered and inspected the strings and the startup sequence to conclude that the code was generated using MSVS 2010 and not MSVS6, as claimed by the Rich headers.

Our Approach. We modify the technique in § 3.1 to detect the list of compilers involved in building a Windows binary. We use IDA Pro [22] to disassemble the Windows binary files. For this specific application, we increased the number of features to the 3,000 most frequent for each type since we are now classifying entire binaries rather than functions. We then augment the classifier to predict the list of compiler IDs used to build a binary, and compare the predicted IDs with the list of compiler IDs included in the binary's Rich header. Our current prototype includes two naïve approaches for making comparisons. First, we include a conservative approach that flags binaries whose Rich header includes compiler IDs that our classifier does not predict. This approach allows the classifier to include false positives. Alternatively, we include a strict comparison mode that flags any binary whose Rich header does not exactly match the predicted compiler IDs. We acknowledge that these approaches are not robust against adversarial attacks.

5.1.1 Evaluation. Since we lack ground truth for OlympicDestroyer, we used the projects in Corpus IV to build several versions of the binaries. We used three different versions of the Visual Studio compiler (VS2017, VS2019, and VS2022) and two compiler optimization options, namely the `debug` and `release`. In total, each project had six sets of binaries corresponding to the compiler versions and optimization options. The compilers build programs in C and C++ languages and link static libraries built by other compilers. Each language compiler has a unique compiler ID that the binary's Rich header documents. Furthermore, the statically linked libraries have their own Rich headers that compilers append to the final binary's Rich header. There are 14 different compiler IDs in total.

We set aside 10% of the binaries from Corpus IV as a validation set. We split the remaining binaries into a training set (75%) and a testing set (25%) for each compiler ID. We balance the data following the same procedure described in Section 3.1, where the training and testing set of each compiler ID contains the same amount of positive and negative samples.

5.1.2 Findings. Table 5 shows the result of compiler ID identification. On average, the classifier achieves an F-1 score of 97.2%. Furthermore, it attains a perfect score (100%) for the Visual Studio 2022 (VS2022) compiler family. We examined the feature importance to gain insights into why the classifier performed well. An analysis of the top features revealed that different minor versions of the same compiler share common code patterns. For example, four minor versions of the VS2022 compiler, `01047c4f`, `01057c4f`, `01047cc1`, and `01057cc1`, all share the same feature `callq *%rax | addq #IMM# %rsp` in their list of top 15 features, while no other compilers have this feature. Some features also appear in almost all compilers after a specific version. For example, the feature `movq`

Compiler ID	Compiler Name	Training Samples	Testing Samples	F-1 (%)
1046852	[C] VS2017	4636	1546	96.5
1056852	[C++] VS2017	4635	1545	95.9
1047552	[C] VS2019	4635	1545	96.2
1057552	[C++] VS2019	4633	1545	96.5
01047c4f	[C] VS2022	4633	1545	100
01057c4f	[C++] VS2022	4632	1544	100
010469a8	[C] Unknown	3729	1243	94.5
01047cc1	[C] VS2022	3703	1235	98.7
010475c2	[C] VS2019	3688	1230	94.8
010575c2	[C++] VS2019	1336	446	95.7
01057cc1	[C++] VS2022	1326	442	96.1
010569a8	[C++] Unknown	1297	433	96.5
0104784b	[C] VS2022	1249	417	100
0105784b	[C++] VS2022	474	158	100
Average		3186	1062	97.2

Table 5: Classification result on compiler ID identification.

`#IMM# #MEM# %rip | movq #IMM# #MEM# %rip` is among the top 15 features for nine compilers for VS2019 and VS2022.

Next, to assess how well this classifier can be used to verify the validity of information contained in the header, we simulated a scenario where a binary's Rich header may contain fraudulent information. We use the verification set for this experiment. We divide these binaries into two groups to generate the positive and negative sets. For binaries in the positive set, we randomly shuffle parts of the Rich header information so that it contains incorrect compiler IDs; for binaries in the negative set, we keep their Rich header unchanged.

Under the conservative comparison configuration, the verifier achieves an F-1 score of 95.8%. Out of the 1032 binaries, the classifier correctly verifies 989 of them. For cases where it fails, 30 are false positives (i.e., the verifier incorrectly flags the Rich header as tampered), and 13 are false negatives (i.e, the Rich headers are tampered with, but the verifier fails to detect that). Using the strict comparison setting, the F-1 score drops to 86.5%. In this case, it correctly verified 891 binaries while inducing 137 false positives and four false negatives. The increase in false positive cases is expected: the classifier needs improvement to distinguish between the C/C++ compilers of the same version. For the false negatives, the claimed Rich header includes one additional compiler where its C/C++ compiler counterpart of the same version exists in the correct Rich header. We posit that the F-1-score could be improved using the decomposition technique discussed next.

5.2 Grouping Third-Party Libraries

Binary decomposition of programs is becoming increasingly important as software programs grow larger and more complex — *e.g.,* often importing code from many third-party libraries to accomplish some needed functionality. Binary decomposition is the process of dissecting a binary into groups, where each group consists of a set of related functions from the same library or program. This capability is helpful for both developers and end-users alike. Analysts could decompose the binary for benign programs for third-party library detection, enabling downstream tasks such as license violation detection and vulnerability detection. Similarly, developers can use the extracted information to decide if to include a third-party library in their build process. In the case of malicious binaries, binary decomposition could help forensic analysts find correlated groups of functions that provide more context for attribution and malware behavior analysis.

Towards these objectives, several proposals for binary decomposition have been suggested. Karande et al. [26], for example, decompose a C++ binary into groups where each group consists of functions in the same C++ class. By contrast, the approach of Yang et al. [61] (coined ModX) breaks a binary into groups of functions where each group consists of functions that are related to each other (*e.g.,* together, they perform a specific task) and come from the same library or program source. ModX utilizes a community detection algorithm to dissect the call graph of a binary into groups. However, using these techniques to assist us with our forensic evaluations of complex binaries, we found that the commonly used algorithms provided very different groupings. There was no immediate way to access the quality of their respective output.

After inspecting a handful of binaries that use external libraries, we noticed that certain functions in the same library seemed to have similar sets of optimization passes applied, likely due to being built by the same compiler version with similar compiler flags. Based on that observation, we designed a method for improving the decomposition process by instead using compiler pass information as a stopping criterion for the iterative Girvan-Newman (GN) community detection algorithm [35] used by ModX [61].

Our Approach. We designed and implemented a prototype that accepts a binary file, extracts the call graph, recovers the fine-grained compiler pass information, and then uses the GN community detection algorithm to infer groupings of functions. The GN algorithm operates in rounds where it iteratively dissects the call graph of a binary into groups of related functions and continues doing so until each function forms its own group. We modified the algorithm to use compiler pass information to select the best result among all its iterations.

To generate call graphs, we simply use `objdump` to disassemble the binary into functions and look for direct call instructions in each function. For compiler pass information recovery, we use the technique described in Section 3 to identify the list of compiler passes applied to each function.

To determine the best result across all iterations, we use a metric based on the similarity of the compiler passes applied to the functions in the same group. If the similarity is higher than a pre-defined threshold, we consider the grouping robust (i.e., self-consistent). We examine thresholds ranging from `0.1` to `0.9`, with a step size of `0.1`. To generate an overall score, we measure the ratio of the number of groups that meet the threshold divided by the total number of groups. We compare the difference in the score for each round and select the round with the highest difference.

5.2.1 Evaluation. To collect ground truth, we built a custom program that gives us full control of the libraries used and precise knowledge of the source of each function. Our program is a TCP server program that encrypts and decrypts messages with a symmetric cipher. The program uses `libhydrogen` [12] for its encryption and decryption operations and `libc` for threading, string operations, and networking. Internally, the libraries also use functions from `libgcc` and `libgcc_eh` and a single function from `libclang_rt`. We compile our program and the `libhydrogen` library statically. For other libraries, we use the default version in Fedora Linux 37.

To obtain a mapping of function names and their corresponding libraries, we use `objdump` to disassemble the static library file (`.a` file) of `libhydrogen` and all static library files in `/usr/lib` and `/usr/lib64` to retrieve a list of function names for each library. We use the mapping for ground truth.

We compare our results with three alternative algorithms for extracting communities from large networks: Greedy Modularity Communities [11], Asynchronous Label Propagation [41], and Louvain Community Detection [6]. We chose these algorithms because implementations are readily available, use the same input (i.e., only the call graph) as GN, and support directed graphs. Unlike GN, which outputs results after each round, these algorithms output the best groupings (based on internal metrics) at termination.

Algorithm	Groups	Consistent Groups	Avg. Group Size	Median Group Size
Greedy	28	16	14.81	3
Asynchronous	215	67	4.44	3
Louvain	180	156	5.14	2
Girvan-Newman	**86**	**72**	**9.54**	**5**

Table 6: Comparison of algorithms. A self-consistent group means that all its functions belong to the same library or program source. Avg/median size is the average/median size of self-consistent groups.

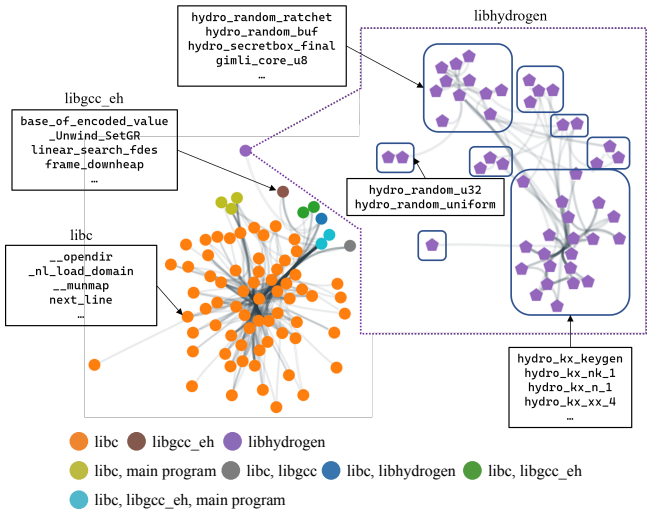

Figure 5: Visualization of decomposed grouping using our modified GN algorithm. Best viewed in color. Each circle represents a group and each pentagon represents a function. Disjointed nodes are not displayed.

5.2.2 Findings. Table 6 summarizes the result with different community detection algorithms. For GN adapted with our selection approach, the best result was achieved at a threshold of `0.6` (Appendix D provides the results at other thresholds). A close examination shows that the Louvain Community Detection and GN outperform the others based on the notion of having many self-consistent groups. For the Louvain Community Detection, 86% of the groups are consistent groups versus 83% for GN. While it may seem that Louvain Community Detection generates a better result, the downside is that it generates small groups. Within the 156 consistent groups for Louvain, the average and median group sizes are only 5.14 and 2 functions, respectively; on the other hand, for the 86 consistent groups found using our modifications to GN, the average and median group sizes are 9.54 and 5 functions. Our conjecture is that for forensics tasks, a smaller number of large groups is more favorable than a larger amount of tiny groups. The rationale is based on the idea that larger groups with more functions can better reveal intra-group code similarity.

Figure 5 visualizes the decomposition using the modified GN algorithm. There, each node is a group, and each edge represents a function call between two groups. Most groups contain only functions from `libc` because it is a large library with multiple submodules. For three of the four libraries, we can generate at least one group where all functions in the group belong to the same library. The right side of that figure shows the intra-group consistency when we zoom into the `libhydrogen` group and decompose it into smaller sub-groups. Manual inspection of the sub-groups shows that the functions match well with their corresponding semantics.

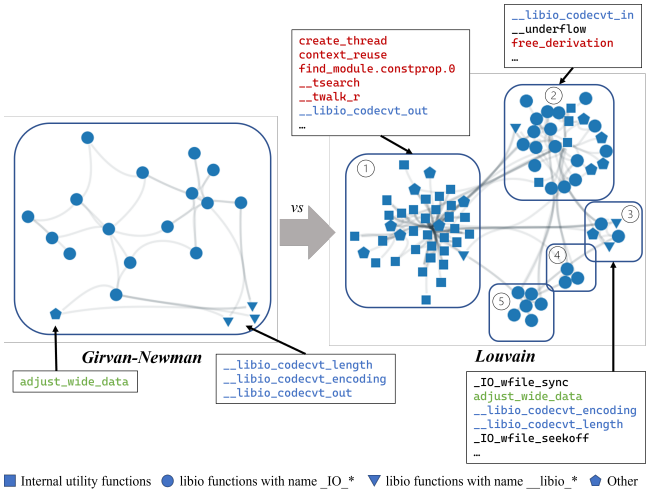

Figure 6: A single group of `libc` functions clustered by GN versus how the same functions were clustered by the Louvain Community Detection algorithm. Each point is a function.

To illustrate the differences in outcomes, Figure 6 shows a group identified by our approach compared to how the Louvain Community Detection algorithm grouped the corresponding functions. Louvain splits the related functions across multiple groups. Some groups are small, while others are large groups that contain unrelated functions (e.g., red elements in Group 1) coming from source files in different sub-directories.

Overall, we posit that by using groupings with less noise, third-party license detectors and vulnerability scanners [50, 64] could become better at identifying statically linked libraries. This would

likely be the case for LibDB [50], which creates function embeddings [60] from call graphs and then uses a nearest neighbor algorithm to select the best match (among an indexed set of libraries) given an input set of clusters decomposed from an unknown binary. Clusters that have many unrelated functions generate low similarity scores, thereby inducing false negatives.

6 DISCUSSION

Our case studies provide an exposition of the diverse security implications arising from compiler optimizations. Our research lays a foundation for effectively utilizing compiler provenance information to enhance numerous security tasks. In particular, developers stand to benefit from a practical tool for identifying and rectifying DSE in privacy-sensitive code. Given the prevalence of potential DSE vulnerabilities in everyday applications, end-users can leverage the provenance model to evaluate potential risks in mission-critical binaries, thus further strengthening system security.

Regarding ROP gadgets, establishing a correlation between gadgets and passes facilitates a streamlined approach to identifying problematic sections of code. Developers can thus focus on targeted fixes only for those sections without being bogged down by the intricacies of low-level gadget finding. In a similar vein, end-users can incorporate a provenance report as part of a software bill of materials (SBOM) to assess risks based on these correlations. Importantly, given that compiler provenance can be obtained solely from binaries, such SBOMs can be generated without relying on vendor support. Moreover, we envision that malicious rich header modifications and binary decomposition, augmented by compiler provenance, can be building blocks for empowering end-users with tools to generate or validate SBOMs from the bottom up.

We hope that the research presented here will serve as a source of inspiration for others to explore paths that can be further investigated, both in terms of refining techniques and identifying new use cases to bolster security tasks. For example, it is clear that the precision of the DSE classifier presented in Section 3 would need to be augmented prior to widespread adoption. One direction is to enhance the classifier with supplementary information obtained through binary decomposition. We anticipate that this approach could improve not only DSE detection but also other compiler provenance tasks. Another possible refinement is improving the granularity of rich header analysis with binary decomposition. With clusters of functions within a binary, the Rich header verification technique presented here could verify each submodule of a binary, further improving the verification accuracy.

7 LIMITATIONS

Our findings are specific to version 14 of the widely popular LLVM compiler. That said, given that we extract the list of passes applied to each compilation unit by instrumenting the `LegacyPassManager` and deduce information regarding removed store instructions by instrumenting the `Dead Store Elimination` pass, we are confident that with additional engineering effort these modifications can be ported to other versions of LLVM. We leave the design and implementation of models that support multiple versions of LLVM, as well as different compilers (e.g., GCC), as an open problem. Similarly, while our study only evaluates X86-64 binaries, we note that most of the optimization passes that we examine are machine-independent passes that operate at the IR level. Hence, our findings should apply to other architectures as well.

Lastly, we do not cover all types of compiler-induced vulnerabilities. Additionally, as modern compilers are continuously developing, our models must be periodically re-trained. Nevertheless, we believe our selection of compiler-induced vulnerabilities aptly demonstrates the severity of compiler-induced security issues observed in the wild. We leave the analysis of other vulnerability types (*e.g.,* constant time violations or those related to orthogonal specification issues) for future work.

8 CONCLUSIONS

We present a large-scale evaluation of fine-grained compiler configuration information recovery and demonstrate three downstream tasks that can benefit from using the recovered artifacts. Overall, our evaluation shows that one can reliably detect the presence of certain passes applied at the level of functions. Our analysis of compiler-induced vulnerabilities on real-world programs reveals that C/C++ projects continue to suffer from persistent state violation caused by the dead store elimination optimization and an increased amount of code reuse gadgets due to optimizations. These violations arise because of implicit specification modifications that do not align with the programmers' intent. The case for Rust is more encouraging. Lastly, we present approaches for forensic tasks, including Windows Rich header verification and improving binary decomposition.

9 AVAILABILITY

To promote research in this area, our extensions and data are available at https://github.com/zeropointdynamics/passtell.

REFERENCES

[1] Saed Alrabaee, Mourad Debbabi, and Lingyu Wang. 2019. On the feasibility of binary authorship characterization. *Digital Investigation* 28 (2019), S3–S11.
[2] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe Rust? *Proceedings of the ACM on Programming Languages* 4 (2020), 1 – 27.
[3] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. RUDRA: finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*. 84–99.
[4] Joseph Biden. 2021. Executive Order 14028: Improving the Nations's Cybersecurity. Federal Register.
[5] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *IEEE Symposium on Security and Privacy*. 227–242.
[6] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* (2008), P10008.
[7] Michael D Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. 2021. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proceedings of the ACM on Programming Languages* 5 (2021), 1–30.
[8] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard E. Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2018. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. In *Annual Network and Distributed System Security Symposium*.
[9] cgzones. 2023. Memory erasure followups. https://github.com/linux-pam/linux-pam/pull/599. Accessed: 2023-09-07.
[10] Yu Chen, Zhiqiang Shi, Hong Li, Weiwei Zhao, Yiliang Liu, and Yuansong Qiao. 2019. HIMALIA: Recovering compiler optimization levels from binaries by deep learning. In *Intelligent Systems and Applications: Proceedings of the 2018 Intelligent Systems Conference Volume 1*. 35–47.
[11] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical review. E, Statistical, nonlinear, and soft matter physics* 70 (2004).

[12] Frank Denis. 2023. LibHydrogen. https://github.com/jedisct1/libhydrogen. Accessed: 2023-04-17.

[13] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The correctness-security gap in compiler optimization. In *IEEE Security and Privacy Workshops*. 73–87.

[14] Yufei Du, Ryan Court, Kevin Snow, and Fabian Monrose. 2022. Automatic Recovery of Fine-grained Compiler Artifacts at the Binary Level. In *USENIX Annual Technical Conference*. 853–868.

[15] Robert J Erbes. 2022. *How to Build SBOM from Binaries: A Round About Story*. Technical Report. Idaho National Lab., Idaho Falls, ID (United States).

[16] GCC Team. 2020. GCC 10 Release Series – Changes, New Features, and Fixes. https://gcc.gnu.org/gcc-10/changes.html. Accessed: 2023-04=27.

[17] GCC Team. 2021. GCC 11 Release Series – Changes, New Features, and Fixes. https://gcc.gnu.org/gcc-11/changes.html. Accessed: 2023-04=27.

[18] Global Research & Analysis Team, Kaspersky Lab. 2018. The devil's in the Rich header. https://securelist.com/the-devils-in-the-rich-header/84348/. Accessed: 2023-03-02.

[19] Google. 2022. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: 2022-08-29.

[20] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *Comput. Surveys* 54, 3, Article 51 (apr 2021).

[21] Xu He, Shu Wang, Yunlong Xing, Pengbin Feng, Haining Wang, Qi Li, Songqing Chen, and Kun Sun. 2022. BinProv: Binary Code Provenance Identification without Disassembly. *International Symposium on Research in Attacks, Intrusions and Defenses* (2022).

[22] Hex Rays. 2023. IDA Pro - Hex Rays. https://hex-rays.com/ida-pro/

[23] Michael J Hohnka, Jodi A Miller, Kenrick M Dacumos, Timothy J Fritton, Julia D Erdley, and Lyle N Long. 2019. Evaluation of compiler-induced vulnerabilities. *Journal of Aerospace Information Systems* 16, 10 (2019), 409–426.

[24] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. 2012. Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In *Workshop on Offensive Technologies*.

[25] JFrog. 2022. Conan, the C/C++ Package Manager. https://conan.io.

[26] Vishal Karande, Swarup Chandra, Zhiqiang Lin, Juan Caballero, Latifur Khan, and Kevin Hamlen. 2018. BCD: Decomposing binary code into components using graph-based clustering. In *Proceedings of the Asia Conference on Computer and Communications Security*. 393–398.

[27] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, Vol. 30.

[28] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. *Proceedings of the Workshop on Programming Languages and Operating Systems* (2017).

[29] Michael Larabel. 2021. LLVM Clang 13 Performance Is In Great Shape For Intel Xeon "Ice Lake". https://www.phoronix.com/review/intel-icelake-clang13. Accessed: 2023-04-27.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*. 75–86.

[31] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *USENIX Security Symposium*. 2099–2116.

[32] Niko Matsakis. 2016. Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html. Accessed: 2023-04-29.

[33] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. *Network and Distributed System Security Symposium* (2022).

[34] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *International Conference on Learning Representations*.

[35] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.

[36] Michalis Papaevripides and Elias Athanasopoulos. 2021. Exploiting Mixed Binaries. *ACM Transactions on Privacy and Security* 24, 2, Article 7 (jan 2021), 29 pages.

[37] Sangdon Park, Xiang Cheng, and Taesoo Kim. 2022. Unsafe's Betrayal: Abusing Unsafe Rust in Binary Reverse Engineering toward Finding Memory-safety Bugs via Machine Learning. *ArXiv* abs/2211.00111 (2022).

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[39] Davide Pizzolotto and Katsuro Inoue. 2021. Identifying Compiler and Optimization Level in Binary Code from Multiple Architectures. *IEEE Access* (2021).

[40] Michal Poslušný and Peter Kálnai. 2020. Rich Headers: Leveraging this mysterious artifact of the PE format. In *Proceedings of the Virus Bulletin Conference*.

[41] Usha Nandini Raghavan, Réka Albert, and Soundar R. T. Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review. E, Statistical, nonlinear, and soft matter physics* 76 (2007).

[42] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. 2015. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation* 14 (2015), S146–S155.

[43] ReFirmLabs. 2021. Binwalk. https://github.com/ReFirmLabs/binwalk. Accessed: 2022-08-29.

[44] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* 15, 1 (2012), 1–34.

[45] Nathan Rosenblum, Xiaojin Zhu, and Barton P Miller. 2011. Who wrote this code? identifying the authors of program binaries. In *European Symposium on Research in Computer Security*. 172–189.

[46] Nathan E Rosenblum, Barton P Miller, and Xiaojin Zhu. 2010. Extracting compiler provenance from program binaries. In *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 21–28.

[47] Sascha Schirra. 2022. Ropper. https://github.com/sashs/Ropper. Accessed: 2022-11-07.

[48] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. In *IEEE European Symposium on Security and Privacy*. 1–15.

[49] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy*. 574–588.

[50] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: An Effective and Efficient Framework for Detecting Third-Party Libraries in Binaries. *International Conference on Mining Software Repositories* (2022), 423–434.

[51] The Rust Team. [n.d.]. Overview of the compiler - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/overview.html. Accessed: 2023-08-04.

[52] Zhenzhou Tian, Yaqian Huang, Borun Xie, Yanping Chen, Lingwei Chen, and Dinghao Wu. 2021. Fine-Grained Compiler Identification With Sequence-Oriented Neural Modeling. *IEEE Access* 9 (2021), 49160–49175.

[53] R. Tsoupidi, R. Castañeda Lozano, E. Troubitsyna, and P. Papadimitratos. 2023. Securing Optimized Code Against Power Side Channels. In *IEEE Computer Security Foundations Symposium*. 242–257.

[54] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Krügel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Network and Distributed System Security Symposium*.

[55] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering* 1 (2016), 236–247.

[56] George D Webster, Bojan Kolosnjaji, Christian von Pentz, Julian Kirsch, Zachary D Hanif, Apostolis Zarras, and Claudia Eckert. 2017. Finding the needle: A study of the PE32 Rich header and respective malware triage. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 119–138.

[57] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147.

[58] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–25.

[59] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *USENIX Security Symposium*.

[60] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[61] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the International Conference on Software Engineering*. 1393–1405.

[62] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead store elimination (still) considered harmful. In *USENIX Security Symposium*. 1025–1040.

[63] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. 2023. Software Bills of Materials Are Required. Are We There Yet? *IEEE Security & Privacy* 21, 2 (2023), 82–88.

[64] Dan Zhang, Ping Luo, Wei Tang, and Min Zhou. 2020. OSLDetector: Identifying Open-Source Libraries through Binary Analysis. In *International Conference on Automated Software Engineering*. 1312–1315.

[65] Tianning Zhang, Miao Cai, Diming Zhang, and Hao Huang. 2022. SeBROP: blind ROP attacks without returns. *Frontiers of Computer Science* 16, 4 (2022), 164818.

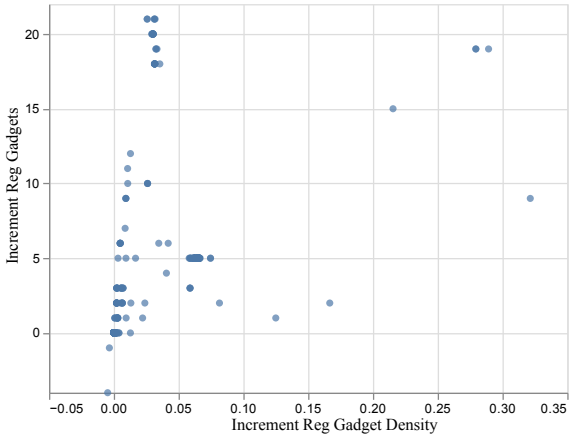

Figure 7: Changes in the number of gadgets that increment a register in binaries in Dataset II after the Control Flow Optimizer pass. The X-axis shows the density of the gadgets. The Y-axis shows the amount of gadgets. Each dot is a binary.

A COMPILER PASS PREDICTION USING DIFFERENT MODELS

Optimization Pass	QDA	NN	LSVM	NB	LGBM
Dead Store Elimination	6.5	47.8	43.6	28.1	**52.0**
Aggressive Dead Code Elimination	1.2	31.5	23.3	37.0	**39.9**
Bit-Tracking Dead Code Elimination	**93.7**	53.1	47.9	40.7	58.5
Remove Dead Machine Instructions	**98.2**	74.3	72.4	62.4	80.1
Early Machine Loop Invariant Code Motion	60.3	**89.9**	87.8	51.9	89.7
Machine Loop Invariant Code Motion	7.4	58.9	59.7	46.6	**69.8**
Loop Invariant Code Motion	47.9	83.6	80.9	42.7	**86.3**
Machine Common Subexpression Elimination	36.9	81.1	76.6	36.8	**81.7**
Early CSE	71.6	87.4	84.4	57.9	**88.7**
Early CSE w/ MemorySSA	52.5	69.8	68.1	47.4	**75.2**
Loop Strength Reduction	40.2	90.4	90.0	37.4	**91.4**
Peephole Optimizations	70.4	93.1	**94.6**	71.3	94.4
SROA	97.8	**99.5**	**99.5**	95.2	99.3
Control Flow Optimizer	87.8	96.8	97.5	85.6	**97.7**

Table 7: Compiler pass prediction result in F-1 score (%) using different machine learning models. QDA: Quadratic Discriminant Analysis; NN: Neural Network; LS: Linear SVM; NB: Naïve Bayes; LGBM: LightGBM

To evaluate the performance of different machine learning models, we tested approaches based on Quadratic Discriminant Analysis (QDA), Neural Networks, Linear SVMs, and Naïve Bayes. We use the same dataset and experimental setup as in Section 3.2, with Corpus I and III as the training set and Corpus II as the testing set. We computed the average across 10 runs for each model. Table 7 shows the F-1 scores of each model for the passes with security-related implications. Since LightGBM performed the best on 9 of the 14 passes it was selected as our classifier of choice. The results also show that an ensemble of techniques could offer a viable path for improving overall performance.

Pass	Feature	Feature Score
Dead Store Elimination	`mul`	70
	`cmp | cmove`	65
	`test %r13b %r13b`	60
Aggressive Dead Code Elimination	`jne #TARGET# | mov #MEM# %rbp %rax`	69
	`idiv`	69
	`pand`	60
Bit-Tracking Dead Code Elimination	`mov #MEM# %rsp %rax | mov %rax %rax`	66
	`movzbl %cl %eax`	52
	`mov #MEM# %rip %r15`	46
Remove Dead Machine Instructions	`cmove | xor`	87
	`xor %edi %edi`	82
	`jmp #TARGET# | lea #MEM# %rip %rdi`	73
Early Machine Loop Invariant Code Motion	`START test %rsi %rsi`	117
	`START vcvtsd2ss %xmm0 %xmm0 %xmm1`	108
	`START test %edx %edx`	99
Machine Loop Invariant Code Motion	`mov #IMM# %edi | mov #IMM# %esi`	77
	`mov %rdx %r15`	65
	`mov %r12 %rdi | xor %esi %esi`	59
Loop Invariant Code Motion	`r11w`	118
	`END cmp #MEM# %rbx %edx`	104
	`fcmovbe`	99
Machine Common Subexpression Elimination	`punpckhqdq`	103
	`call #TARGET# | mov #IMM# %rdx`	89
	`mov #MEM# %rbp %r10`	74
Early CSE	`pop %r13`	83
	`cmp #IMM# %ecx | mov %al #MEM# %rbp`	71
	`mov #MEM# %rbx %r8`	68
Early CSE w/ MemorySSA	`mov #MEM# %rax %r13`	65
	`jle | test`	61
	`movapd | mov`	59
Loop Strength Reduction	`mov #MEM# %r13 %rax`	119
	`sub %eax %ecx`	105
	`mov | movw`	103
Peephole Optimizations	`movsd`	133
	`mov %edx %ecx`	102
	`mov %r15 %rsi`	82
SROA	`START mov #MEM# %rsi %r8`	121
	`START mov %r9 %r10`	118
	`fsub`	109
Control Flow Optimizer	`mov #MEM# %rip %rbp`	144
	`mov #IMM# %eax`	138
	`lea #MEM# %rip %r8 | mov #IMM# %al`	131

Table 8: Top 3 features for each compiler pass listed in Table 3

B FEATURE IMPORTANCE OF SECURITY-RELATED PASSES

Table 8 lists the top three features for the passes listed in Table 3 and their feature importance. Higher feature importance score means that the feature is more definitive for the classifier to make the decision. Features that begin with `START` and `END` are the first and last instruction of a function, respectively.

C IMPACT OF PASSES ON GADGET AVAILABILITY

An example of another pass that increases gadget availability is shown in Figure 7. Here, when the compiler runs the 79th function

Threshold	Groups	Consistent Groups	Avg. Group Size	Median Group Size
0.1-0.3	9	3	2.33	3
0.4	91	77	8.92	5
0.5	70	56	9.51	4
0.6	**86**	**72**	**9.54**	**5**
0.7	180	114	6.25	3
0.8	377	187	3.68	2
0.9	502	201	2.94	3

Table 9: GN algorithm with different thresholds. The threshold selected in Section 5.2.1 is highlighted.

pass, which is the `Control Flow Optimizer` pass for most programs, the number of gadgets that increment a register increased by 3.59 on average. Figure 7 shows the changes of the amount of gadgets that increment a register and the density of this type of gadgets. While a large portion of binaries see no change in the number of register incrementing gadgets, many binaries show increase in both the number of gadgets and the density of the gadgets.

D BINARY DECOMPOSITION USING DIFFERENT THRESHOLDS

Table 9 lists the decomposition statistics using our modified GN algorithm with different thresholds. We selected the decomposition result at threshold 0.6 because the result has the highest average and median group size as well as a high ratio of consistent groups.