Automatic Recovery of Fine-grained Compiler Artifacts at the Binary Level

Yufei Du, *University of North Carolina at Chapel Hill;* Ryan Court and Kevin Snow, *Zeropoint Dynamics;* Fabian Monrose, *University of North Carolina at Chapel Hill*

https://www.usenix.org/conference/atc22/presentation/du

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Automatic Recovery of Fine-grained Compiler Artifacts at the Binary Level

Yufei Du[1], Ryan Court[2], Kevin Snow[2], and Fabian Monrose[1]

[1]University of North Carolina at Chapel Hill
[2]Zeropoint Dynamics

Abstract

Identifying a binary's compiler configuration enables developers and analysts to locate potential security issues caused by optimization side-effects, identify binary clones, and build compatible binary patches. Existing work focuses on identifying compiler family, version and optimization level of a binary using semantic features and deep learning techniques. Unfortunately, in practice, binaries are an amalgamation of objects and functions that can be compiled at different optimization levels with a variety of individual, fine-grained, optimizations that may be applied depending on the structure of the code. Hence, rather than recovering high-level artifacts, i.e., compiler family, version, and optimization level, we explore the recovery of individual, fine-grained, optimization passes for each function in a binary. To do so, we develop an approach using specially crafted features alongside intuitive and understandable machine learning models. Our evaluation on 15 popular open-source repositories shows that our approach compares favorable with the state-of-the-art deep learning approach in compiler family, compiler version and optimization level identification. For fine-grained optimization passes, our evaluation on 149,814 functions from 552 binaries in four popular open-source repositories shows that our approach achieves an average F-1 score of 92.1% for all optimization passes and an average F-1 score of 89.8% for optimization passes that could have negative impacts on security. Moreover, our approach includes experimental support for dynamic feature extraction via binary emulation, and our results shows that such features offer promising potential in improving the accuracy of optimization pass identification.

1 Introduction

Modern compilers serve much more than simple translators that convert human-readable program code to machine instructions. They are also fully automated optimizers that improve the performance of code via numerous translations, including the removal of unused code, re-ordering of instructions, replacing expensive computations with more efficient ones, and merging functions. Ideally, these optimizations do not change the behavior of the program in any way, other than making the resulting code faster and smaller.

However, while compiler optimizations are preformed in ways that should not interfere with the normal execution of a program, some optimizations could have a negative impact on security. In fact, recent studies [3, 19, 21, 22, 24] have shown that certain optimizations could nullify protections and verification of secure functions as well as introduce timing side channels. For example, the dead code elimination optimization could remove instructions that erase sensitive data after using it, causing the sensitive data to be vulnerable to leakage if there is a memory error later on in the program. Similarly, the strength reduction optimization that replaces expensive operations with more efficient ones could open side channels. Developers are usually unaware of these optimizations because the compiler automatically chooses the set of optimizations to apply based on the optimization level, code structure, target architecture, and target processor family. Moreover, even subtle changes can create additional code reuse gadgets, causing the compiled program to be more vulnerable to attacks [1].

One way to avoid these pitfalls is by having developers manually tweak their compilation scripts to avoid applying potentially risky optimizations to secure functions that handle sensitive data. However, for large projects, this would be a daunting and tedious task. In order to maximize performance while avoiding risky optimizations, developers need to manually tweak optimization flags in addition to the optimization level. With hundreds of both architecture-independent and architecture-dependent flags, manually tweaking optimization flags is challenging and time-consuming [5]. Moreover, some compilers also include hidden optimizations that cannot be manually controlled, and the code base for mod-

ern compilers is too large for users to review and study for the logic behind optimizations [7]. In practice, developers for security-critical projects, such as OpenSSL and mbed TLS, take another approach by implementing workarounds in the source code of secure functions to "confuse" the compiler such that it does not apply optimizations to the secure code [19, 21, 24].

These workarounds, however, are not always stable. As compilers introduce more optimizations in each release, a workaround that successfully tricks one version may not be effective for a future version, and developers then need to implement a more complex workaround [19]. Therefore, a solution that helps identify optimizations applied to a binary at the function level could offer significant practical value: developers could use such an approach to, for example, verify that the program is compiled using optimizations that do not negatively impact security before releasing the binary.

In addition to safety verification, compiler optimization classification could be beneficial to binary code clone detection and binary patching. Compiler configurations can cause significant degradation in the performance of clone detection techniques [6, 8, 10, 12]. Similarly, in binary patching [4], locating the vulnerable function to be patched becomes more difficult in the presence of certain compiler optimizations.

Facing some of these challenges ourselves when trying to safely perform binary patching at a function level, we revisited the state of the art in compiler artifact recovery. We found that the most comprehensive of these techniques [2, 16, 17, 20] focus on identifying the compiler family, the major compiler version, and the optimization level for a binary, either for individual functions or for the entire binary — but, we need more detailed information (i.e., the passes that may have been applied). Unfortunately, this level of recovery has not been well explored, and even when it has been mentioned, the authors conclude that "*[f]urthermore, some flags would be challenging, if not impossible to detect, the dead code elimination flag being one example*" [15].

Additionally, we found that contemporary approaches rely on either semantic features (e.g., the control flow graph) or employ deep learning. As the interpretability of the outputs was a key motivating factor for us, we instead opted for the use of shallow learning with specially crafted features. To our surprise, our approach performed on par with or better than the state of the art [20] that uses highly-tuned neural networks for optimization level identification (e.g, `-O1` versus `-O3`). More importantly, we take a step further and show that contrary to recent statements by Pizzolotto and Inoue [15], one can detect the application of certain optimization passes with good accuracy. Our approach, coined *PassTell*, helps identify optimization passes that affect security for individual functions, such as different forms of dead code elimination, code motion, and strength reduction passes.

Our specific contributions include:

1. We designed PassTell, a new approach in compiler configuration identification that recovers the optimization passes applied at the function level.
2. We explored the effects of using dynamic features extracted by force-executing each function. We show that the use of such features offers potential in improving accuracy, albeit in certain cases.
3. We evaluated our machine learning approach and compared the results with the state-of-the-art. We find that our approach performs on par with the state-of-the-art in identifying compiler family, compiler version, and coarse-grained optimization level.
4. We evaluated our approach using four variants of 138 programs from four open source repositories built with the latest development version of the Clang 14 compiler. Our approach is capable of identifying most optimization passes with high accuracy.

2 Background

2.1 Compiler Optimization

Modern compilers (e.g., GCC, LLVM, and ICC) offer complex optimizations that improve the performance and reduce the code size of the compiled program. In theory, these compilers provide different levels of optimizations, and programmers only need to specify an optimization level for the compiler to automatically apply the corresponding set of optimizations.

In practice, however, optimization is more complicated than simply applying a fixed set of optimizations passes for each optimization level. The LLVM compiler [11], for example, uses pass managers to control the passes to apply as well as the order of running the passes. The pass manager considers multiple factors when deciding the passes to run in addition to the optimization level specified by the user, including the target architecture, the target processor generation, and the source code structure. For example, for programs targeting outdated x86 processors, the compiler would avoid applying optimizations that use the AVX instructions that were recently introduced, and for functions without loops, the pass manager would avoid running optimizations that improve loop performance. Therefore, knowing the optimization level of a binary is *not* enough to determine the exact set of optimizations applied to the binary.

2.2 Security Implications

While compilers can take care to ensure that their optimizations do not change the behavior of normal program execution, fully automated reasoning about the purpose of deliberately ineffective or seemingly useless operations added by the developers is not (yet) possible. Hence, such instructions are targets for optimization, potentially undermining security assumptions. D'Silva et al. [3] called this problem the "correctness-security gap" and defined three types of security violations caused by compiler optimization: persistent state, side channel attacks, and undefined behavior.

A persistent state violation is when data persists outside of the scope it is designed to be available. D'Silva et al. [3] listed three optimizations that could cause this violation: dead code elimination, function inlining, and code motion. For example, in a password verification function where the password is temporarily stored in the memory during verification, the compiler may consider the operations that erase the local memory to be dead code and remove them, causing the password to exist in the memory after it is used, until it is eventually overwritten by a later function. Similarly, if a trusted security-sensitive function is inlined in an untrusted function, then the lifetime of the local variables of the trusted function would be extended to when the untrusted function returns. Finally, code motion may switch the order of instructions to avoid unnecessary computation or to improve locality. This optimization may cause the program to write sensitive values to memory before verifying that the operation is needed. Beside these three optimizations, Simon et al. [19] added that in situations where the entire stack frame needs to be erased, any optimization that changes the size or the layout of the stack frame such as tail-call optimizations may cause the erasure to be incomplete.

Compiler optimizations could also introduce side-channels that leak information about the program's execution based on its timing or memory usage. To avoid side-channels, the developer may add unnecessary or inefficient operations to functions, but the optimizations may simplify or remove these operations, thereby reintroducing the side-channel. D'Silva et al. [3] listed three optimizations that could introduce side-channels: common subexpression elimination, which merges multiple instructions into one instruction to avoid duplicate computation; strength reduction, which replaces expensive instructions with more efficient ones, and peephole optimization, which inspects surrounding instructions to find opportunities to reorder or replace instructions for simpler computation or better locality.

Our work focuses on identifying optimizations that could cause persistent state violations or side-channels. Identifying undefined behavior (i.e, violations caused by undefined behavior when developers use semantics that are undefined by the specification) is out of scope.

3 Related Work

Rosenblum et al. [18] presented seminal work in the area of compiler identification from binary files. Their approach focuses on identifying only the compiler family for code snippets in the IA-32 architecture using a probabilistic graphical model. Later on, Rosenblum et al. [17] extended that work and presented Origin, a tool that identifies compiler family, compiler version, and optimization level for each function in a binary. Origin uses a linear support vector machine model with features including idioms of instructions, sub-graphs of the control-flow graph, and high-level layout of functions such as the starting address. However, for optimization level, Origin could only perform coarse-grained identification with two options: "low" for `-O0`, `-O1`, and "high" for `-O2`, `-O3`.

A few years later, Rahimian et al. [16] presented a different approach (called BinComp) for compiler provenance identification. BinComp focuses on identifying the compiler family, compiler version, and optimization level for the entire binary. Different from Origin, BinComp heavily utilizes features extracted from utility functions added by the compiler to identify the compiler version and optimization level. These utility functions include program initialization, the startup code, and the termination code. While these functions could be highly indicative of an optimization level, it is impossible to perform identification for each function in a binary. Therefore, BinComp could only identify the compiler configuration used to compile the main routine of a program.

More recent work [2, 15, 20, 23] in compiler provenance identification started using neural networks for classification. Chen et al. [2] presented HIMALIA, a classifier using recurrent neural network to identify the optimization level for each function of a binary. HIMALIA uses vectors of disassembled instructions as features and uses two recurrent neural networks for classification. One network classifies the function into one of the four classes: `-O0`, `-O1`, `-O2/O3`, and `-Os`; the other network then differentiates `-O2` and `-O3`. While the evaluation of HIMALIA includes binaries compiled with different versions of the LLVM Clang compiler, it only focuses on identifying the optimization level of each function, making no distinction of optimizations applied in different compiler versions. Yang et al. [23] presented BinEye, a classifier using convolutional neural network to identify optimization levels for each object in ARM binaries. Since each instruction in the ARM architecture is four bytes, BinEye uses the first 1024 instructions of each object as raw features and extracts word and position embeddings from them. Tian et al. [20] presented NeuralCI,

a classifier with either convolutional neural network or recurrent neural network to identify the compiler family, compiler version, and optimization level for each function in a binary. NeuralCI uses Word2Vec [14] embedding to allow instructions with variable size. Similar to BinEye, the evaluation of NeuralCI combined `-O2` and `-O3` into one coarse-grained optimization level of `OH`.

Most recently, Pizzolotto and Inoue [15] presented an approach that uses either a convolutional neural network or a long-short term memory network to identify the compiler family and optimization level for code snippets of 2KB in seven different architectures. This approach includes either the raw bytes or the opcodes as features but concluded that raw bytes lead to better results when large amounts of training data are available. Similar to HIMALIA, the evaluation of this approach includes five different optimization levels: `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`. As NeuralCI performed the most in-depth and realistic evaluation, and it was shown to outperform the other approaches that classify at the function level, we select it for comparison later on in this paper.

4 Approach

We now present PassTell, an approach for identifying the set of optimization passes likely applied to each function in a binary file. Figure 1 shows the overall workflow.

4.1 Dataset Generation

Our dataset used to train the classifier includes functions compiled with different optimizations. Each function includes a set of optimization passes that were applied during compilation and the instructions in the function. The overall workflow of dataset generation is as follows: first, we compile programs with different optimization levels and record the list of optimizations applied to each function; then, we disassemble the binary to retrieve the instructions of each function; finally, we sanitize the instructions by removing detailed memory addresses, call targets, and immediate values.

To train and evaluate our classifier, we first need to gather the optimization passes that modify a function during compilation. While the Clang frontend of the LLVM compiler [11] has an option to list the optimizations applied to each function during compilation, this option outputs all passes that run, even the ones that make no modification. Therefore, we made modifications to extract the optimization passes that modify a function during compilation. Section 5 discusses the modifications we made to the LLVM compiler.

After disassembling the binary file, we sanitize the instructions before feature extraction. Specifically, we replace all memory addresses with the `#MEM#` label, all call targets with `#TARGET#`, and all immediate values with `#IMM#`. We make this adjustment because the detailed memory addresses, immediate values, and call targets are highly variable across different programs and are not useful in optimization classification. Tian et al. [20] applied similar rules to the instructions.

4.2 Dynamic Feature Generation

As an extension, we also present a method to extract changes of register values (recovered via emulation) and use these dynamic features in our classifier. At present, the dynamic features only include register deltas. We use a binary emulation library to attempt to force-execute each function in the dataset. After the execution of each instruction, we record the address of the instruction, the registers changed, and the deltas of their values. Within these three types of data, the instruction address is only used to compute the coverage of the force-execution and to avoid endless loops.

While it may seem unnecessary to include dynamic features as the dataset already includes the entire disassembly of the function, we posit that dynamic features could still contribute to the classification because some changes in the registers are implicit and not shown in the disassembly. For example, the `FPSW` register includes flags, the stack address, and the current code for floating point operations. Additionally, floating point operations may cause this register to change implicitly. Our main goal of including dynamic features is to explore their potential to improve classification accuracy. We expect progress can still be made in future work.

4.3 Feature Extraction

Category	Feature Type	Example
Static	Opcode	`call`
	Instruction	`mov esi ecx`
	Register	`rsi`
	2-gram of opcodes	`pop | ret`
	2-gram of instructions	`pop rbp | ret`
	First instruction	`push r15`
	Last instruction	`xchg ax ax`
Dynamic	Register value delta	`rbp=-248`

Table 1: Feature types and examples for each type of feature used in our approach

By default, we use seven types of static features: opcode, instruction, register, two-gram of opcodes, two-gram of instructions, and the first and last instruction of a function. The register value delta is an optional feature. Table 1 lists an example of each feature type.

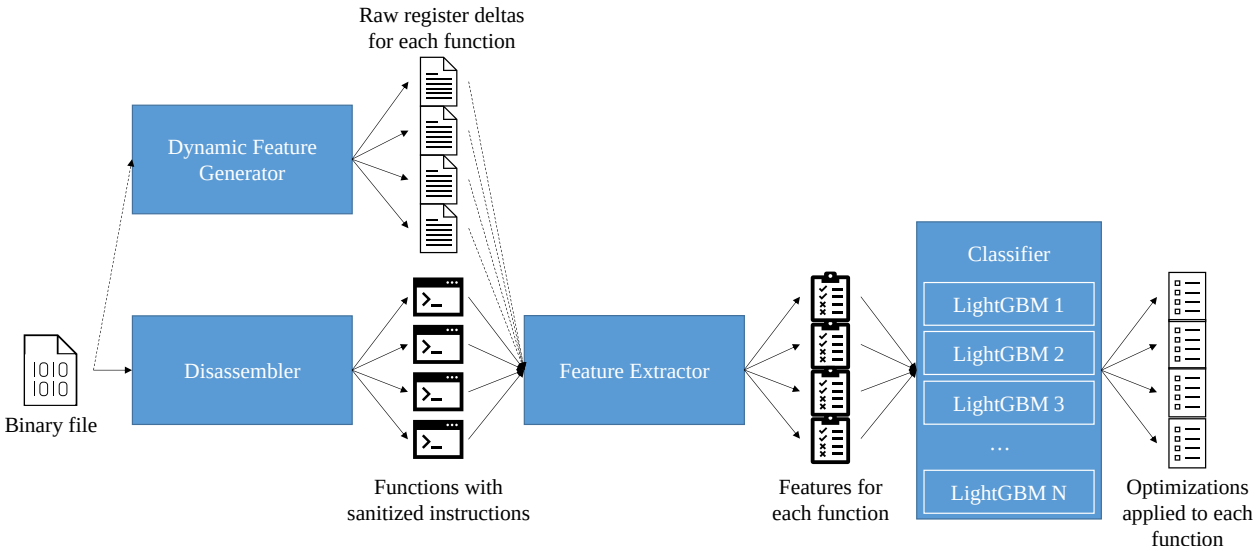

Figure 1: Workflow of PassTell for optimization pass classification

We apply feature selection before extracting features. There are overwhelming amount of different instructions, two-grams of opcodes, two-grams of instructions, and changes of register values, so we select 1,000 features from each of these four feature types, for each optimization pass. This means that we use different features for the classification of each optimization pass. Our feature selection strategy for each optimization pass is as follows: first, we filter the dataset such that it is balanced for the optimization pass (i.e., the number of functions with the optimization and the number of functions without the optimization are the same); second, we sort the features by the number of functions in the balanced dataset that have at least one occurrence of the feature; finally, we select the 1,000 most frequent features of each type. Table 2 shows an example of the selected features for the `Early CSE` pass. For the other feature types, including opcode, register, the first instruction, and the last instruction, we use all features in these types without applying feature selection because there are less features in these types.

All features hold binary values. That is, we check whether the function has this feature or not. For example, a leaf function (i.e., a function that does not call any other functions) that performs arithmetic operations in a loop should have value `0` for the opcode feature `call` because it does not include any call, and it may have value `1` for the opcode feature `test` because it may use the `test` instruction to determine the end condition of the loop.

We decide to use this simple set of binary features as a result of interpretable feature engineering. Our approach with interpretable feature importance allows us to compare and select the most efficient and effective feature types. We also tested using frequencies as features instead of binaries but found minimal improvements.

4.4 Classification

After generating the feature set, we then train our classifier for optimization classification. For each optimization, we train a binary LightGBM classifier [9] that decides if a function is modified by this single optimization.

LightGBM is an implementation of gradient boosting decision tree. We select this classifier instead of deep learning techniques for two reasons. First, modern compilers include a large amount of passes. In our dataset, for example, we observed a total of 83 unique compiler passes applied to functions or components inside a function (e.g., a loop). Each pass requires a separate binary classifier. Therefore, in order to train the dataset with a reasonably large dataset, our choice of classifier should scale well in both training time and memory consumption. Second, classifiers such as support vector machines and neural networks cannot easily demonstrate the reasoning or the importance of features. Decision tree-based classifiers, on the other hand, can more readily show the reasoning and the importance of features.

In the training phase, a list of all optimization passes is generated and a model for each is created. That is, for each function, we create a list of binary labels for each pass to indicate if the function was modified by each of these corresponding passes, then a LightBGM-based classifier is trained for each of those binary labels (i.e., 83 classifiers in total) using the extracted features. For classification, the feature extractor sends each function's features to each of the 83 trained models. The trained model for each optimization pass then decides if that optimization was applied to (and modified) each function, and finally the list of all applied optimization passes is returned for each individual function in the binary.

Static						Dynamic	
Instruction		2-gram of Opcodes		2-gram of Instructions		Register Value Delta	
Feature	Count	Feature	Count	Feature	Count	Feature	Count
`ret`	2,626	`mov; mov`	2,577	`pop rbp; ret`	1,517	`ip=3`	2,710
`call #T#`	2,300	`pop; ret`	2,413	`push rbp; mov rsp rbp`	1,059	`rip=3`	2,710
`add #I# rsp`	1,688	`push; mov`	2,069	`add #I# rsp; pop rbx`	841	`eip=3`	2,710
`push rbp`	1,572	`mov; call`	2,059	`mov rsp rbp; sub #I# rsp`	814	`rsp=-8`	2,582
`pop rbp`	1,527	`call; mov`	1,680	`add #I# rsp; pop rbp`	788	`spl=-8`	2,582

Table 2: Top five selected features for feature types with feature selection for optimization pass `Early CSE` with a sample set of 2,780 functions, including 1,390 positive samples and 1,390 negative samples. (Sanitized keywords such as `#TARGET#` are abbreviated to only the first letter.)

5 Implementation

We now discuss the implementation details of the various components shown in Figure 1.

Dataset Generation We implemented the dataset generation component in Python. The technique takes a source code repository and compiles it with four levels of optimizations (`-O0`, `-O1`, `-O2`, and `-O3`). During compilation, it extracts the ground truth of optimization passes that modified each function from the compiler log. After compilation, `objdump` is used to disassemble the binary in order to retrieve the instructions of each function. Instructions are sanitized by removing detailed memory addresses, call targets, and immediate values (as discussed in Section 4.1).

We modified the legacy pass manager of the LLVM compiler [11] in order to retrieve the list of optimization passes that modifies a function. We made modifications to the latest development version of LLVM 14 at the time of this writing (commit #`c59ebe4`). We added an option to the existing output flag of LLVM's legacy pass manager[1] to list the optimization passes that modify a function or components inside a function (e.g., a loop).

Finally, since we use `objdump` to disassemble the compiled binary files into functions, our tool enables debugging symbols during compilation. However, this is not a hard requirement for classification as one could use external tools such as IDA Pro to determine function boundaries of stripped binaries, but prior efforts found little difference in outcome between the two tactics [20].

Dynamic Feature Generation As mentioned in Section 4.2, PassTell also supports register values extracted during execution to complement static features. To do this, we make use of Zelos [25], a python-based binary emulator platform that supports x86 (both 32 and 64-bit), ARM and MIPS architecture emulation. Under the hood, Zelos makes use of QEMU CPU emulation and implements system call emulation similar to QEMU usermode, but CPU and syscall-level hooks make comprehensive binary instrumentation readily available. Using this tool, we instrument forced emulation of each function in a binary and record the changes to register values after each instruction within function boundaries. To do so, we extended the emulator to execute a binary, then wait until it has mapped itself in memory and pause execution. Then, for each function, the instruction pointer is adjusted to the function start address before execution resumes. While executing the function, `call` instructions are skipped to ensure that recorded register changes reflect only the target function, while also avoiding recursion and potentially long call chains. Before emulating each function, we map a page of memory and fill it with the start address of the region. The address of this region is used as the return address for the target function as well as for all existing register values that appear to be pointers to memory, to avoid errors related to reading or writing unmapped memory.

Feature Extraction and Classification As discussed in Section 4.3, our approach performs feature selection for each optimization pass. As such, having a dedicated feature extraction component that saves all features to files is highly inefficient. Therefore, we combined the feature extraction phase for both static and dynamic features and the classifier into one classifier component.

We implemented our classifier using Python and the LightGBM library [13]. The classifier iterates through each optimization pass, selecting and extracting features and then creating a `LGBMClassifier`. When training, the classifier first filters the dataset such that the dataset is balanced, with the same amount of positive and negative data. Then, the classifier selects the most popular

[1] Recent versions of the LLVM compiler contains two pass managers, and by default, the new pass manager is responsible for all optimization passes before code generation. However, the new pass manager does not contain any utility functions to extract the pass names. Therefore, we use the flag "-flegacy-pass-manager" to use the legacy pass manager for all passes, including the ones before code generation.

features as described in Section 4.3, extracts both static and dynamic features, and trains the `LGBMClassifier`. When classifying, the classifier extracts the static and dynamic features and uses the `LGBMClassifier` to predict whether the function is modified by this optimization pass. After the classifier extracts features and performs classification for all optimization passes, it then merges the results together to generate the final result, the list of optimizations applied to the function.

6 Evaluation

The evaluation includes experiments in two directions. First, we evaluate the effectiveness of our features and our classifier. In this experiment, we compare PassTell with NeuralCI [20], a state-of-the-art approach to compiler configuration identification. To ensure a fair comparison, we first modified our classifier into a multi-class classifier to identify the same compiler configuration as NeuralCI, including the compiler family, the major compiler version, and the optimization level, where each combination is a class. Later, we evaluate our approach in identifying the individual optimization passes. Overall, our experiments seek to answer the following questions:

RQ1 *How does our approach compare to the state-of-the-art in compiler configuration recovery?*

RQ2 *Can our approach provide meaningful information regarding feature significance?*

RQ3 *How well can we infer individual optimization passes?*

RQ4 *Does the inclusion of dynamic features help with classifying individual passes?*

6.1 Compiler Configuration Identification

We use the same benchmark programs as in NeuralCI [20]. To replicate NeuralCI's experimental setup, we combine binaries compiled with `-O2` and `-O3` optimization levels into one class, `-OH`. As our prototype utilizes `objdump` to generate the disassembly for each function, we only use the unstripped binaries. Tian et al. [20] observed no difference in classification performance between the stripped and unstripped binaries. Our dataset thus consist of all dynamically linked unstripped executables from the dataset, including `binutils`, `busybox`, `coreutils`, `curl`, `ffmpeg`, `git`, `gsl`, `libpng`, `openssl`, `postgresql`, `sqlite`, `valgrind`, `vim`, `zlib`, and `gdb`.

While inspecting the dataset, we discovered that the dataset is highly unbalanced, with some configurations having significantly less amount of samples than others. We note that this issue is not limited to NeuralCI, as other approaches [2, 15] also do not balance their datasets. To circumvent this issue, we randomly dropped functions from certain configurations to ensure that all configurations have the same number of functions. In the end, our dataset consists of 4,400 functions for each configuration. We split the dataset into an 80% training set and a 20% testing set. Appendix A describes the issue in detail and includes a comparison of the results of NeuralCI before and after balancing the dataset. Finally, NeuralCI includes only unique functions in its dataset, so functions that are identical across configurations are removed. We apply the same procedure.

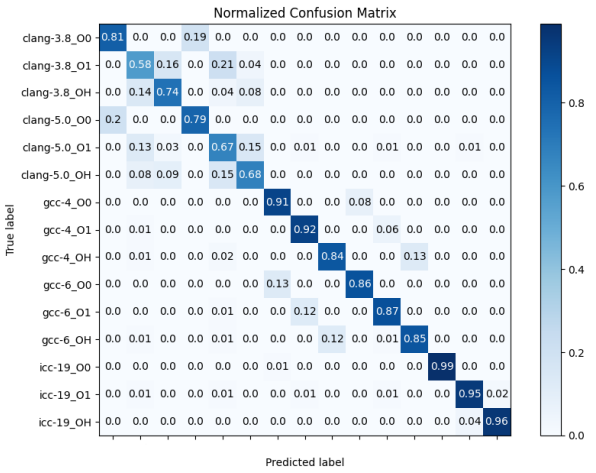

Figure 2: Confusion matrix of our LightGBM model applied to the NeuralCI formulation of the compiler classification problem

Since Tian et al. [20] do not include all the code used to construct features from the dataset, we re-implemented the extraction and abstraction of functions. For extraction, we use `objdump` instead of IDA Pro to parse the body of each function. The abstraction for each function is the same as done by Tian et al. [20], namely, mnemonics and register operands are unchanged, base memory addresses in operands are replaced with the symbol #MEM#, and immediate values are replaced with the symbol #IMM#.

Experiment Results

For the identification of compiler family, compiler version, and optimization level, NeuralCI achieves an average F-1 score of 76.6%, and our approach achieves an average F-1 score of 83.2%. Our re-implementation of NeuralCI produces lower results as reported in their paper [20]. We attribute the variation to be due to the correctly balanced dataset. Overall, the results show that our approach performs better than NeuralCI in identifying the compiler family, the major compiler version, and optimization level.

Interestingly, both the confusion matrix of our ap-

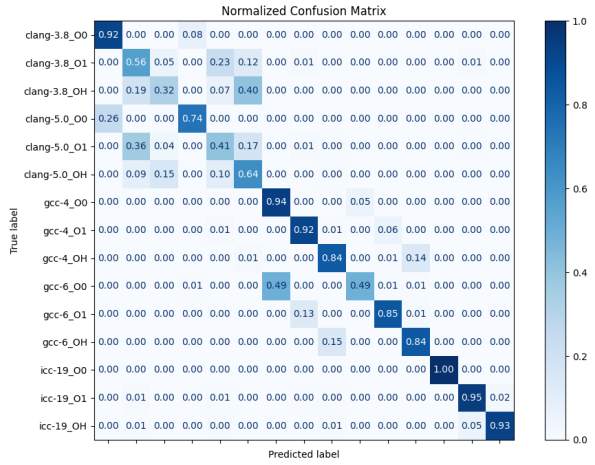

Figure 3: Confusion matrix of NeuralCI in classifying the compiler family, compiler version, and optimization level

proach (Figure 2) and the confusion matrix of NeuralCI (Figure 3) show that identifying the optimization level and the compiler version of binaries compiled with Clang is more challenging. NeuralCI reports similar findings in its evaluation [20]. For functions built by GCC and ICC, both approaches achieve high accuracy in identifying the compiler family, compiler version, and optimization level. For this reason and the fact that we modified the LLVM pass manager to extract pass information, we focus on Clang[2] for the remaining experiments.

6.2 Optimization Pass Identification

Satisfied with the performance and simplicity offered by PassTell, we focused on tackling the more difficult cases with Clang. Specifically, the compiler pass dataset consists of functions from `binutils` (2.37), `coreutils` (9.0), `httpd` (2.4.51), and `sqlite` (3.36.0) programs compiled with Clang 14, using each of `-O0`, `-O1`, `-O2`, and `-O3` optimization levels, generating a total of 149,814 functions in 552 binaries. Then, we balance the dataset for each pass: for each pass, we randomly select an equal amount of functions with the pass applied (i.e., positive samples) and functions without the pass applied (i.e., negative samples). We also limit the maximum number of samples for each pass to 5,000 positive samples and 5,000 negative samples.

Experiment Results

Overall, our approach achieves an average F-1 score of 92.1%. All but three of the 83 passes have an F-1 score higher than 80%, and the three exceptions all have insufficient amount of samples (<150 functions). If we only consider the 73 passes that contain more than 500 samples, the average F-1 score improves to 93.7%. Table 3 depicts the results of our approach using only static features. For brevity, we only list the detailed results of 13 of the 83 passes in total. We pick these 13 passes because they are optimizations that could affect security [3, 19, 21, 24]: dead store elimination, dead code elimination, code motion, tail call optimization, common subexpression elimination, strength reduction, and peephole optimizations. For these passes, our approach achieves an average F-1 score of 89.8%. The findings further show that, contrary to Pizzolotto and Inoue [15]'s statement, even passes that seem unlikely to be detected, such as dead code elimination, can be identified with high accuracy.

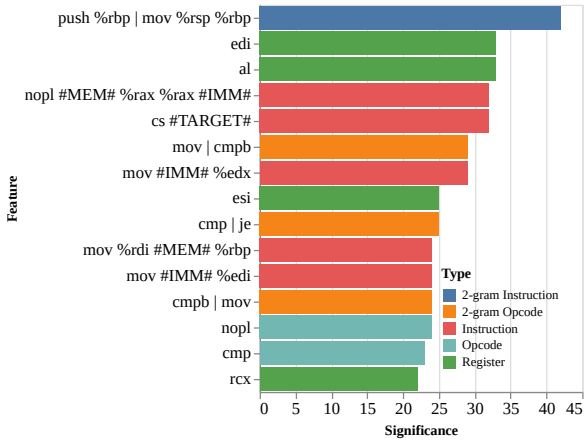

Figure 4: Top 15 Features for `Aggressive Dead Code Elimination`

To understand why our approach works as well as it does, we inspected the feature significance of some of the security-affecting passes. We choose to inspect the feature significance for `Aggressive Dead Code Elimination` and `Peephole Optimizations` because these two passes offer sufficient descriptions about their purposes in the code comment of the LLVM compiler's source code. Figure 4 shows the top 15 features for the `Aggressive Dead Code Elimination` pass and the feature type of each feature. The description of the optimization pass in the LLVM compiler's source code indicates that this pass considers all code to be dead unless proven otherwise and removes all the dead instructions, especially dead code involving loops. The top feature in this case checks whether the function updates the function pointer. Clang omits updating the frame pointer on optimization levels above `-O1`. Similarly, the instruction features `nopl #MEM# %rax $rax #IMM#` and `cs #TARGET#`[3] are instructions padding instructions without any semantic meaning whose purpose

[2]Clang is a front-end of LLVM and is part of the LLVM infrastructure.

[3]This feature is actually a `nopw` NOP instruction. Due to the different format `objdump` uses for instructions involving the `cs` segment register, this instruction is not parsed correctly. Since segment registers

Pass	Training Samples	Testing Samples	Precision (%)	Recall (%)	F-1 (%)
Dead Store Elimination	1332	444	86.3	85.8	85.7
Aggressive Dead Code Elimination	1092	364	83.9	83.5	83.4
Bit-Tracking Dead Code Elimination	2512	838	87.6	87.5	87.5
Remove dead machine instructions	7500	2500	88.7	88.6	88.6
Early Machine Loop Invariant Code Motion	7500	2500	93.4	93.3	93.3
Machine Loop Invariant Code Motion	739	247	89.4	89.0	89.0
Loop Invariant Code Motion	7500	2500	90.8	90.6	90.6
Tail Call Elimination	88	30	86.6	86.6	86.6
Machine Common Subexpression Elimination	7500	2500	88.5	88.1	88.1
Early CSE	7500	2500	92.5	92.2	92.2
Early CSE w/ MemorySSA	7500	2500	88.9	88.6	88.6
Loop Strength Reduction	7500	2500	95.4	95.4	95.4
Peephole Optimizations	7500	2500	98.0	98.0	98.0
Average			90.0	89.8	89.8

Table 3: Precision, recall, and F-1 results on security-related passes when using static features.

is to enforce a 16-byte alignment between functions, and Clang only adds these instructions at `-O1` or above. Since `Aggressive Dead Code Elimination` is never applied at `-O0`, these three features effectively remove functions with `-O0` optimization level.

Four other features among the top 15 features include various forms of the `comp` compare instruction, which commonly appears in loops. This finding matches the description of the pass. Finally, some of the top features show significance in the usage of certain registers. Since this pass is applied before register allocation in the compiler pipeline, we speculate that the removal of dead instructions reduces the amount of required registers, causing the register allocator to not use certain registers.

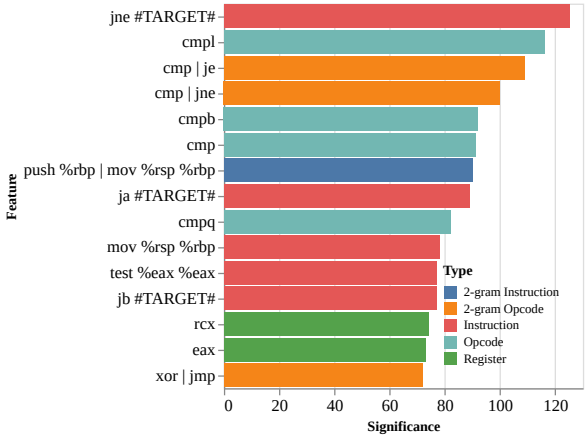

Figure 5: Top 15 Features for `Peephole Optimizations`

The top features for `Peephole Optimizations` show similar patterns. Figure 5 shows the top 15 features. The description in the code comments suggests that this pass performs four types of optimizations: optimization of sign/zero extension instructions, optimization of comparison instructions, optimization of loads, and optimization of copies and bitcasts. The top features show that our classifier mainly capture the second type of optimizations that optimizes comparison instructions. Six of the 15 features include a variant of the `cmp` compare instruction; four other features include a jump instruction that usually follows a compare instruction, and one feature includes a `test` instruction, which is functionally similar to a compare instruction. Similar to `Aggressive Dead Code Elimination`, the top features for `Peephole Optimizations` also include the feature that saves the frame pointer in order to detect `-O0` functions.

6.3 Case Study on Optimization-induced Vulnerabilities

Although prior studies [3, 19, 21, 22, 24] have shown that compiler optimizations can weaken protections put in place by safe coding practices, it remained unclear whether the nullification of protections could in fact lead to information leakage or other attacks for real-world programs. Thus, we studied three real-world programs and examined how protections introduced by programmers are affected by the dead store elimination optimization: `BusyBox` (1.35.0), `httpd` (2.4.52), and `crypto++` (5.6.4).

BusyBox `BusyBox` is a popular embedded program that combines many UNIX utilities into a single binary. Some of the utilities included in `BusyBox` require password encryption or authentication, such as the `passwd` utility, the `cryptpw` utility, the built-in HTTP server, and the built-in FTP server. Therefore, `BusyBox`'s codebase includes hashing functions such as MD5 and SHA. When a utility needs to encrypt a password, the utility would call

are rarely used in 64-bit x86 programs, and we did not find any instructions utilizing that register other than the padding NOP, we conclude that this small implementation quirk does not impact our results.

pw_encrypt, which then calls the corresponding MD5, SHA, or DES encryption functions. The SHA encryption function, sha_crypt, uses both a local stack object, L, and heap objects, key_data and salt_data, to store intermediate values during the encryption. Listing 1 shows a snippet of this function. Before the function returns, it attempts to erase the three objects using memset. However, at -O3 optimization level, the dead store elimination optimization removes all three calls to memset. Thus, the intermediate values remain in the memory after the function returns even though developers took the necessary precautions to erase the data.

```
static char * NOINLINE sha_crypt(/*const*/
    char *key_data, /*const*/ char *salt_data)
{
    ...
    struct {
        ...
    } L __attribute__((__aligned__(__alignof__
        (uint64_t))));
    ...
    salt_data = xstrndup(salt_data, salt_len);
    ...
    key_data = xstrdup(key_data);
    ...
    /* Clear the buffer for the intermediate
        result so that people
       attaching to processes or reading core
           dumps cannot get any
       information.  */
    memset(&L, 0, sizeof(L));
    memset(key_data, 0, key_len);
    memset(salt_data, 0, salt_len);
    free(key_data);
    free(salt_data);
    ...
    return result;
}
```

Listing 1: Code snippet from the sha_crypt function

We tested this program using its cryptpw utility that prints the hashed password in the format of Linux's passwd format from a given password in plain text. The cryptpw utility calls pw_encrypt to hash the password text, which calls sha_crypt if SHA mode is selected. At -O0 optimization level, all three objects that contains intermediate values in sha_crypt (L, key_data and salt_data) are overwritten to 0 properly at the end of the function. At -O3 optimization level, however, the entire value of L is left on the stack at the end of the function. For the heap objects, the calls to free overwrite the first 16 bytes of both key_data and salt_data. Since salt_data is less than 16 bytes, its value cannot be recovered. However, in situations where the input plain text password is longer than 16 characters, the size of key_data would be larger than 16 bytes. In this case, part of the intermediate value stored in key_data would persist in the heap memory after its scope. After sha_crypt returns, pw_encrypt then calls a simple clean up function and returns to the caller utility. At this point, both the stack object L and the partial leftover from the heap object key_data still exist in the memory without being overwritten. This means that if an attacker finds a memory disclosure vulnerability in the caller utility of BusyBox before the intermediate values are eventually overwritten by subsequent functions, then the attacker can recover the entire value of L and/or the value of key_data after the first 16 bytes. Since BusyBox includes an HTTP server and an FTP server that both use pw_encrypt, it would be possible for an attacker to launch an attack remotely.

We have submitted a bug report for this issue [4].

```
int get_password(struct passwd_ctx *ctx)
{
    char buf[MAX_STRING_LEN + 1];
    ...
    else {
        ...
        apr_password_get("Re-type new password
            : ", buf, &bufsize);
        if (strcmp(ctx->passwd, buf) != 0) {
            ctx->errstr = "password
                verification error";
            memset(ctx->passwd, '\0', strlen(
                ctx->passwd));
            memset(buf, '\0', sizeof(buf));
            return ERR_PWMISMATCH;
        }
    }
    memset(buf, '\0', sizeof(buf));
    return 0;
    ...
}
```

Listing 2: Code snippet from the get_password function

Httpd and Crypto++ The HTTP server httpd contains a support utility program, htpasswd, that manages the files that store usernames and passwords. This program includes a function get_password() that reads the password entered by the user to a local buffer, and stores the buffer to a passwd_ctx struct. Before it returns, the function calls memset to erase the buffer such that the password entered by the user would not stay in the memory. Listing 2 shows a snippet of this function. Before any return statement, the function erases the memory of the buffer, buf, that contains the user-entered password at line 11 and line 15 in Listsing 2. However, at -O3 optimization level, the dead store elimination optimization removes the function call to memset that erases buf, causing the password in plain text to persist in the stack memory after get_password() returns, contrary to the developers' intention.

We tested this program and discovered that immediately after the function returns, we could recover the exact password in plain text in the stack memory. It is

[4] https://bugs.busybox.net/show_bug.cgi?id=14806

worth noting that after `get_password` returns to its caller, `mkhash`, the caller later calls other functions, overwriting the stack memory that contains the password as the stack frame of `get_password` is smaller than the stack frame of subsequent functions. Therefore, if an attacker finds a memory disclosure vulnerability in `mkhash` between the call to `get_password` and the subsequent function call, then they can retrieve the password in plain text.

`Crypto++`, a cryptography library written in C++, contains a similar issue as `htpasswd`. The function `CAST256::Base::UncheckedSetKey` uses a call to `memset` to erase its local variable, `kappa`, which holds the hashed key. At `-O3` optimization, the dead store elimination optimization removes this call, causing the secret key to remain in stack memory after the function returns.

For validation purposes, we use the entire compiler pass dataset (discussed in Section 6.2) as our training set for `BusyBox` and `crypto++`. For `httpd`, our compiler pass dataset also contains `httpd`, albeit a different minor version, so we exclude all functions in `httpd` from our training set. Our classifier correctly identifies the `Dead Store Elimination` pass for all three functions at `-O3` optimization level. At `-O0`, the classifier mis-identifies `sha_crypt` for `BusyBox` but identifies the missing of the pass correctly for `httpd` and `crypto++`.

6.4 The Effects of the Dynamic Features

Finally, we evaluate the value of including dynamic features. For that, we use the same dataset as in Section 6.2, but apply additional filtering. Specifically, because our current implementation of our dynamic feature generator is unable to generate register features for all functions in the dataset, we filter the dataset to only include functions that our generator achieved a minimum coverage threshold. Additionally, we only include passes that were applied to more than 500 functions. Since our goal is to evaluate the effects of the dynamic features and not the coverage of the feature generator itself, we view this as a reasonable way to understand what impact dynamic features could have on classification performance.

To compare to the baseline, we ran our approach using both static features and dynamic features. For the latter, we experimented with coverage thresholds ranging from 30% to 60%. For these configurations, we used the same filtered dataset and artificially removed dynamic features to reach the target coverage.

Experiment Results

Table 4 depicts the results. We only show cases where the difference in F-1 score is greater than 1%. While the average F-1 score appears similar regardless of the inclusion of dynamic features, the detailed results tell a more compelling story. Some passes, such as `Remove dead machine instructions`, show a notable decrease in F-1 score, while others (e.g., `Early CSE w/ MemorySSA`) improve by almost 4.0%.

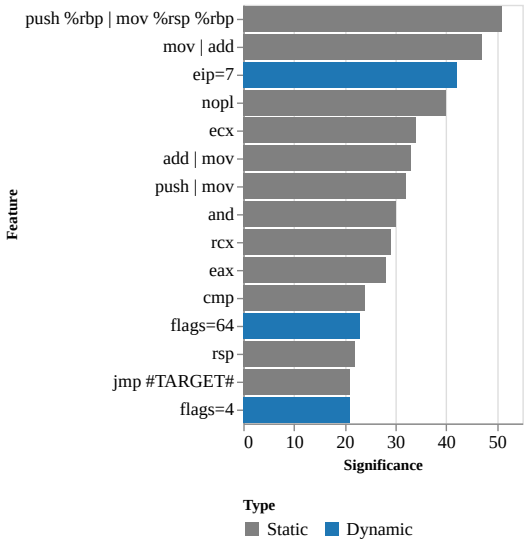

Figure 6: Top 15 Features for `Early CSE w/ MemorySSA` with both static and dynamic features

To better understand the reasons, we examined the most important featured as deemed by the `LGBMClassifier` classifier. Figure 6 shows the top 15 features and the significance of each feature for the `Early CSE w/ MemorySSA` pass. The description of this pass indicates that this pass removes "trivially redundant instructions". The top 15 features include three dynamic features: `eip=7`, `flags=64`, and `flags=4`. The `EIP` register is the instruction pointer, and the delta of this register shows information of instruction size, which is not available in the static features. Similarly, the `FLAGS` register, as partial alias of the `EFLAGS` register, contains various processor flags that are implicitly set during arithmetic operations and interrupts. Since this register is only set implicitly as a side effect of instructions, static approaches cannot extract this information by analyzing the disassembly or the binary code.

An examination of the `Rotate Loops` pass yields similar insights. Here, the top 15 features (Figure 7) include two dynamic features: `flags=-68` and `eip=10`. These results indicate that dynamic features, instruction pointer and processor flags in particular, can provide useful information in the detection of some optimization passes.

Lastly, we noticed that as we varied the coverage threshold, a few passes showed a noticeable improvement in F-1 score at higher coverage levels. For these passes, the result appears to be related to the number of dynamic features and their significance. For example, for `Early CSE w/ MemorySSA`, at 30% coverage, the top 25

Pass	Training Samples	Testing Samples	Static Only (%)	Static & Dynamic (%)
Early CSE w/ MemorySSA	607	203	82.2	86.1
Rotate Loops	400	134	84.2	86.5
Merge disjoint stack slots	598	200	94.4	96.5
Control Flow Optimizer	2,200	734	95.6	97.1
Canonicalize natural loops	396	132	87.9	89.4
Peephole Optimizations	1,297	433	94.9	96.3
Two-Address instruction pass	3,454	1,152	94.9	96.0
Remove Redundant DEBUG_VALUE analysis	1,515	505	88.1	87.1
PostRA Machine Sink	634	212	90.1	89.1
Live DEBUG_VALUE analysis	3,030	1,010	97.4	95.9
Machine Instruction Scheduler	1,303	435	93.3	91.2
Simplify the CFG	1,912	638	91.2	89.0
Remove dead machine instructions	903	301	91.0	86.7
Average for All Passes with at Least 500 Samples			92.9	93.0

Table 4: F-1 scores using only static versus static and dynamic features. The table list only results where the difference of F-1 score is >= 1%. The coverage threshold is set to >= 70%. Security-related passes are highlighted in dark grey.

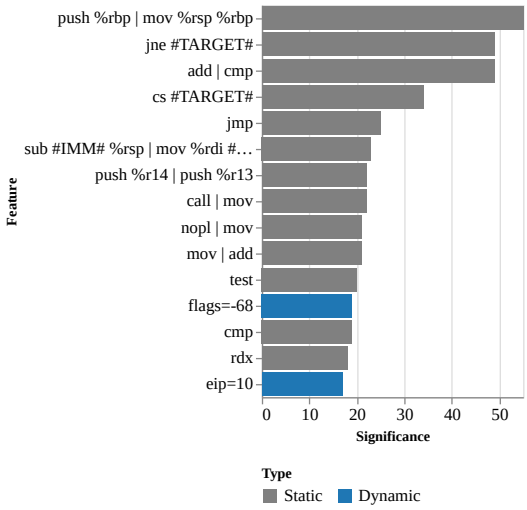

Figure 7: Top 15 Features for `Rotate Loops` with both static and dynamic features

features include four dynamic features, and at 70% or higher, the top 25 features include eight dynamic features. We posit that future improvements in the way dynamic features are collected could boost the classification for more passes, without negatively impacting others.

7 Limitations

Our implementation of the data collection component only records optimization passes applied to functions or components within a function (e.g., loops and basic blocks). Optimization passes applied to larger units such as modules and call graphs are not studied. Therefore, our analysis does not cover cross-function optimizations such as the function inlining. This limitation stems from the fact that the LLVM compiler does not report the specific functions modified by a module pass or call graph pass. Similarly, our data collection component does not support extracting whole-program optimizations applied at link time, which could also negatively impact security [21].

In addition, our approach targets only binary files directly generated by the compiler. Thus, it can not be used in situations where the binaries are modified after compilation, such as obfuscated binaries or binaries with binary patches applied. This limitation is not unique to us.

8 Conclusion

We presented a light-weighted approach to the problem of compiler configuration identification. Our approach combines a novel technique for feature extraction and a scalable classifier for performing compiler provenance recovery. To further improve the accuracy of our classifier, we explored the use of dynamic features extracted by force-executing functions using a binary emulator. Overall, our approach shows comparable results as the state-of-the-art in the original problem of identifying the compiler family, the compiler version, and the optimization level, and pushes the field forward by showing that one can even identify individual optimization passes.

9 Availability

Our coarse-grained compiler configuration classifier and our fine-grained compiler pass classifier are available at `https://github.com/zeropointdynamics/passtell`. The balanced compiler configuration dataset (§6.1), the compiler pass dataset (§6.2), and the compiler pass dataset with high dynamic feature coverage (§6.4) are also available. Furthermore, Zelos, our binary emulator we use to generate dynamic features, is open sourced [25].

References

[1] M. D. Brown, M. Pruett, R. Bigelow, G. Mururu, and S. Pande. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *ACM Conference on Programming Languages*, 5:1–30, 2021.

[2] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao. Himalia: Recovering compiler optimization levels from binaries by deep learning. In *IntelliSys*, 2018.

[3] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *IEEE Security and Privacy Workshops*, pages 73–87, 2015.

[4] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee. Automating patching of vulnerable open-source software versions in application binaries. In *NDSS*, 2019.

[5] K. Georgiou, Z. Chamski, A. A. García, D. May, and K. I. Eder. Lost in translation: Exposing hidden compiler optimization opportunities. *ArXiv*, abs/1903.11397, 2019.

[6] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Conference on Mining Software Repositories*, pages 63–72, 2011.

[7] M. J. Hohnka, J. A. Miller, K. M. Dacumos, T. J. Fritton, J. D. Erdley, and L. N. Long. Evaluation of compiler-induced vulnerabilities. *Journal of Aerospace Information Systems*, 16(10):409–426, 2019.

[8] Y. Hu, Y. Zhang, J. Li, and D. Gu. Binary code clone detection across architectures and compiling configurations. In *International Conference on Program Comprehension*, pages 88–98, 2017.

[9] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, 2017.

[10] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned, 2021.

[11] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, page 75, USA, 2004.

[12] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.

[13] Microsoft Corporation. Light gradient boosting machine. URL https://github.com/microsoft/LightGBM.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[15] D. Pizzolotto and K. Inoue. Identifying compiler and optimization level in binary code from multiple architectures. *IEEE Access*, 2021.

[16] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.

[17] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *International Symposium on Software Testing and Analysis*, pages 100–110, 2011.

[18] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–28, 2010.

[19] L. Simon, D. Chisnall, and R. Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *IEEE European Symposium on Security and Privacy*, pages 1–15, 2018.

[20] Z. Tian, Y. Huang, B. Xie, Y. Chen, L. Chen, and D. Wu. Fine-grained compiler identification with sequence-oriented neural modeling. *IEEE Access*, 9:49160–49175, 2021.

[21] A. Venkatesh, A. B. Handadi, and M. Mory. Security implications of compiler optimizations on cryptography–a review. *arXiv preprint arXiv:1907.02530*, 2019.

[22] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM Symposium on Operating Systems Principles*, pages 260–275, 2013.

[23] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun. Understand code style: Efficient CNN-based compiler optimization recognition system. In *IEEE Conference on Communications*, pages 1–6, 2019.

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

[25] Zeropoint Dynamics. Zelos, 2020. URL https://github.com/zeropointdynamics/zelos.

A The Unbalanced Dataset

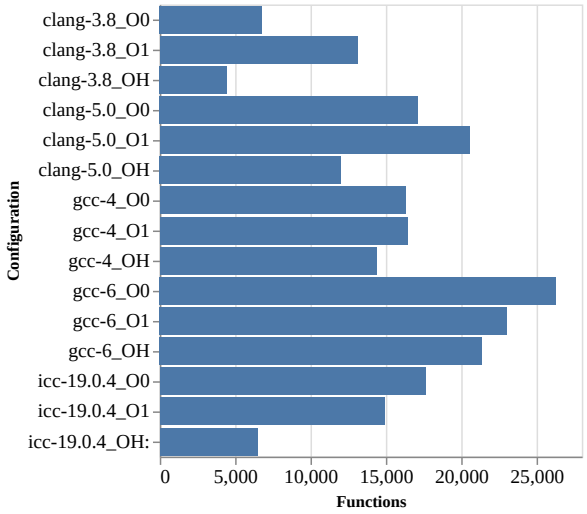

Figure 8: The distribution of samples in compiler configurations in the NeuralCI dataset (prior to re-balancing).

Figure 8 shows the number of functions for each configuration in the original dataset of NeuralCI [20], for 64-bit dynamically linked and unstripped executables. Some configurations, such as Clang 3.8 at `-OH` optimization level, contain significantly less functions than others. This unbalanced dataset could potentially cause bias in evaluation. Therefore, we balanced this dataset by randomly removing functions such that all configurations have the same amount of functions (see Section 6.1).

Dataset	Precision	Recall	F-1
Unbalanced	83.5%	83.5%	83.5%
Balanced	76.6%	76.6%	76.6%

Table 5: Comparison of NeuralCI results using the unbalanced dataset and the balanced dataset

To replicate the evaluation of NeuralCI as accurately as possible, we ran an additional experiment using NeuralCI with the unbalanced dataset. Table 5 shows the results of NeuralCI using the unbalanced dataset and the dataset after we balanced the data size. The results using the unbalanced dataset show roughly the same results as reported by Tian et al. [20], with a negligible variation (<1%). Notice, however, that after balancing the dataset, NeuralCI's performance declines. Compared to the confusion matrix of NeuralCI using our balanced dataset (Figure 3), the confusion matrix of NeuralCI using the unbalanced dataset (Figure 8) shows drastically better results for identifying the GCC version at `-O0` optimization level, likely because GCC 6 has significantly more samples than GCC 4 at `-O0` optimization level. Likewise, the results for Clang functions also differ.

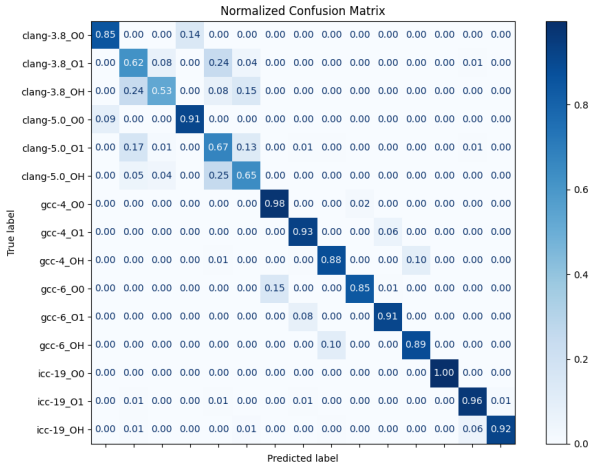

Figure 9: Confusion matrix of NeuralCI using the unbalanced dataset

B Artifact Appendix

Abstract

Our artifacts include the dataset for our experiment in Section 6.1, 6.2 and 6.4, our coarse-grained compiler configuration classifier for Section 6.1, and our fine-grained compiler pass classifier for Section 6.2 and Section 6.4. Our artifacts require a Linux machine (or Windows Subsystem for Linux) with 32GB of RAM and 16GB of storage. Since our classifiers use only shallow learning, a discrete GPU is not required. On our machine with an AMD Ryzen 7 3700X processor, the coarse-grained classifier requires about two hours to finish, and the fine-grained classifier takes about an hour.

Scope

The artifacts allow reproducing our quantitative experiments in Section 6, including coarse-grained compiler configuration identification (Section 6.1), optimization pass identification using only static features (Section 6.2),

and optimization pass identification using both static and dynamic features (Section 6.4).

Contents

Our artifacts include the following contents:

1. `balanced_dataset.csv`: The dataset for coarse-grained compiler configuration classification. As discussed in Section 6.1, this dataset is a balanced subset of the dataset used in NeuralCI [20].
2. `config_classifier.py`: The coarse-grained compiler configuration classifier.
3. `data.csv`: The dataset for fine-grained compiler pass classification used in Section 6.2.
4. `data_dynamic.csv`: The dataset for dynamic feature evaluation used in Section 6.4. As discussed in Section 6.4, this dataset is a subset of `data.csv` that only includes functions whose dynamic feature coverage are at least 70%.
5. `LICENSE`: The license of our artifacts.
6. `passtell.py`: The fine-grained compiler pass classifier.
7. `README.md`: Installation and running instructions.
8. `requirements.txt`: List of required Python libraries.
9. `static_opcode_features.py`: Library module required for `passtell.py`.

Hosting

The classifiers and the datasets are available at https://github.com/zeropointdynamics/passtell in the `main` branch with commit ID `0c88e8d`.

Requirements

Our classifiers have the following requirements:

1. A 64-bit Linux machine with at least 32GB of RAM and 16GB of storage. We have tested our artifacts on Arch Linux (rolling release, updated in May 2022) and Ubuntu 20.04 (Windows Subsystem for Linux).
2. Python 3. For Ubuntu and other Linux distributions that do not have a default `python` command, setting the symbolic link from `python` to `python3` is required. On Ubuntu, this can be done by installing the `python-is-python3` package.
3. Graphviz.
4. Python libraries listed in `requirements.txt`.