Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned

Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim

Abstract—Binary code similarity analysis (BCSA) is widely used for diverse security applications, including plagiarism detection, software license violation detection, and vulnerability discovery. Despite the surging research interest in BCSA, it is significantly challenging to perform new research in this field for several reasons. First, most existing approaches focus only on the end results, namely, increasing the success rate of BCSA, by adopting uninterpretable machine learning. Moreover, they utilize their own benchmark, sharing neither the source code nor the entire dataset. Finally, researchers often use different terminologies or even use the same technique without citing the previous literature properly, which makes it difficult to reproduce or extend previous work. To address these problems, we take a step back from the mainstream and contemplate fundamental research questions for BCSA. Why does a certain technique or a certain feature show better results than the others? Specifically, we conduct the first systematic study on the basic features used in BCSA by leveraging interpretable feature engineering on a large-scale benchmark. Our study reveals various useful insights on BCSA. For example, we show that a simple interpretable model with a few basic features can achieve a comparable result to that of recent deep learning-based approaches. Furthermore, we show that the way we compile binaries or the correctness of underlying binary analysis tools can significantly affect the performance of BCSA. Lastly, we make all our source code and benchmark public and suggest future directions in this field to help further research.

Index Terms—Binary code similarity analysis, similarity measures, feature evaluation and selection, benchmark

1 Introduction

PROGRAMMERS reuse existing code to build new software. It is common practice for them to find the source code from another project and repurpose that code for their own needs [1]. Inexperienced developers even copy and paste code samples from the Internet to ease the development process.

This trend has deep implications for software security and privacy. When a programmer takes a copy of a buggy function from an existing project, the bug will remain intact even after the original developer has fixed it. Furthermore, if a developer in a commercial software company inadvertently uses library code from an open-source project, the company can be accused of violating an open-source license such as the GNU General Public License (GPL) [2].

Unfortunately, detecting such problems from binary code using a similarity analysis is *not* straightforward, particularly when the source code is not available. This is because binary code lacks high-level abstractions, such as data types and functions. For example, it is not obvious from binary code to determine whether a memory cell represents an integer, a string, or another data type. Moreover, identifying precise function boundaries is radically challenging in the first place [3], [4].

Therefore, measuring the similarity between binaries has been an essential research topic in many areas, such as malware detection [5], [6], plagiarism detection [7], [8], authorship identification [9], and vulnerability discovery [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21].

However, despite the surging research interest in binary code similarity analysis (BCSA), we found that it is still significantly challenging to conduct new research on this field for several reasons.

First, most of the methods focus only on the end results without considering the precise reasoning behind their approaches. For instance, during our literature study in the field, we observed that there is a prominent research trend in applying BCSA techniques to cross-architecture and cross-compiler binaries of the same program [11], [12], [13], [15], [16], [19], [22]. Those approaches aim to measure the similarity between two or more seemingly distinct binaries generated from different compilers targeting different instruction sets. To achieve this, multiple approaches have devised complex analyses based on machine learning to extract the semantics of the binaries, assuming that their semantics should not change across compilers nor target architectures. However, none of the existing approaches clearly justifies the necessity of such complex semantics-based analyses. One may imagine that a compiler may generate structurally similar binaries for different architectures, even though they

• *The authors are with KAIST, Daejeon 34141, South Korea. E-mail: {dkay, hahah, sangkilc, sl.son, yongdaek}@kaist.ac.kr.*

Manuscript received 21 November 2020; revised 22 February 2022; accepted 26 June 2022. Date of publication 1 July 2022; date of current version 18 April 2023.
This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant, Korea Government (MSIT) under Grant 2021-0-01332, Developing Next-Generation Binary Decompiler. (Corresponding author: Sang Kil Cha.)
Recommended for acceptance by X. Zhang.
This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2022.3187689, provided by the authors.
Digital Object Identifier no. 10.1109/TSE.2022.3187689

are syntactically different. Do compilers and architectures really matter for BCSA in this regard? Unfortunately, it is difficult to answer this question because most of the existing approaches leverage *uninterpretable* machine learning techniques [12], [13], [19], [20], [21], [23], [24], [25], [26], [27], [28], [29]. Further, it is not even clear why a BCSA algorithm works only on some benchmarks and not on others.

Second, every existing paper on BCSA that we studied utilizes its own benchmark to evaluate the proposed technique, which makes it difficult to compare the approaches with one another. Moreover, reproducing the previous results is often infeasible because most researchers reveal neither their source code nor their dataset. Only 10 of the 43 papers that we studied fully released their source code, and *only two* of them opened their entire dataset.

Finally, researchers in this field do not use unified terminologies and often miss out on critical citations that have appeared in top-tier venues of other fields. Some of them even mistakenly use the same technique without citing the previous literature properly. These observations motivate one of our research goals, which is to summarize and review widely adopted techniques in this field, particularly in terms of generating features.

To address these problems, we take a step back from the mainstream and contemplate fundamental research questions for BCSA. As the first step, we precisely define the terminologies and categorize the features used in the previous literature to unify terminologies and build knowledge bases for BCSA. We then construct a comprehensive and reproducible benchmark for BCSA to help researchers extend and evaluate their approaches easily. Lastly, we design an interpretable feature engineering model and conduct a series of experiments to investigate the influence of compilers, their options, and their target architectures on the syntactic and structural features of the resulting binaries.

Our benchmark, which we refer to as BinKit, encompasses various existing benchmarks. It is generated by using major compiler options and targets, which include 8 architectures, 9 different compilers, 5 optimization levels, as well as various other compiler flags. BinKit contains 243,128 distinct binaries and 36,256,322 functions built for 1,352 different combinations of compiler options, on 51 real-world software packages. We also provide an automated script that helps extend BinKit to handle different architectures or compiler versions. We believe this is critical because it is not easy to modify or extend previous benchmarks, despite us having their source codes. Cross-compiling software packages using various compiler options is challenging because of numerous environmental issues. To the best of our knowledge, BinKit is the first *reproducible* and *extensible* benchmark for BCSA.

With our benchmark, we perform a series of rigorous studies on how the way of compilation can affect the resulting binaries in terms of their syntactic and structural shapes. To this end, we design a simple *interpretable* BCSA model, which essentially computes relative differences between BCSA feature values. We then build a BCSA tool that we call TikNib, which employs our interpretable model. With TikNib, we found several misconceptions in the field of BCSA as well as novel insights for future research as follows.

First, the current research trend in BCSA is founded on a rather exaggerated assumption: binaries are radically different across architectures, compiler types, or compiler versions. However, our study shows that this is not necessarily the case. For example, we demonstrate that simple numeric features, such as the number of incoming/outgoing calls in a function, are largely similar across binaries compiled for different architectures. We also present other elementary features that are robust across compiler types, compiler versions, and even intra-procedural obfuscation. With these findings, we show that TikNib with those simple features can achieve comparable accuracy to that of the state-of-the-art BCSA tools, such as VulSeeker, which relies on a complex deep learning-based model.

Second, most researchers focus on vectorizing features from binaries, but not on recovering lost information during the compilation, such as variable types. However, our experimental results suggest that focusing on the latter can be highly effective for BCSA. Specifically, we show that TikNib with recovered type information achieves an accuracy of over 99% on all our benchmarks, which was indeed the best result compared to all the existing tools we studied. This result highlights that recovering type information from binaries can be as critical as developing a novel machine learning algorithm for BCSA.

Finally, the interpretability of the model helps advance the field by deeply understanding BCSA results. For example, we present several practical issues in the underlying binary analysis tool, i.e., IDA Pro, which is used by TikNib, and discuss how such errors can affect the performance of BCSA. Since our benchmark has the ground truth and our tool employs an interpretable model, we were able to easily pinpoint those fundamental issues, which will eventually benefit binary analysis tools and the entire field of binary analysis.

*Contribution.*In summary, our contributions are as follows:

- We study the features and benchmarks used in the past literature regarding BCSA and clarify less-explored research questions in this field.
- We propose BinKit,[1] the first reproducible and expandable BCSA benchmark. It contains 243,128 binaries and 36,256,322 functions compiled for 1,352 distinct combinations of compilers, compiler options, and target architectures.
- We develop a BCSA tool, TikNib,[2] which employs a simple interpretable model. We demonstrate that TikNib can achieve an accuracy comparable to that of a state-of-the-art deep learning-based tool. We believe this will serve as a baseline to evaluate future research in this field.
- We investigate the efficacy of basic BCSA features with TikNib on our benchmark and unveil several misconceptions and novel insights.
- We make our source code, benchmark, and experimental data publicly available to support open science.

2 Binary Code Similarity Analysis

Binary Code Similarity Analysis (BCSA) is the process of identifying whether two given code snippets have similar

1. https://github.com/SoftSec-KAIST/binkit
2. https://github.com/SoftSec-KAIST/tiknib

semantics. Typically, it takes in two code snippets as input and returns a similarity score ranging from 0 to 1, where 0 indicates the two snippets are completely different, and 1 means that they are equivalent. The input code snippet can be a function [11], [16], [19], [21], [24], [30], [31], [32], or even an entire binary image [7], [8]. Additionally, the actual comparison can be based on functions, even if the inputs are entire binary images [12], [13], [15], [23], [33], [34], [35].

At a high level, BCSA performs four major steps as described below:

*(S1) Syntactic Analysis.*Given a binary code snippet, one parses the code to obtain a disassembly or an Abstract Syntax Tree (AST) of the code, which is often referred to as an Intermediate Representation (IR) [36]. This step corresponds to the syntax analysis in traditional compiler theory, where source code is parsed down to an AST. If the input code is an entire binary file, we first parse it based on its file format and split it into sections.

(S2) Structural Analysis. This step analyzes and recovers the control structures inherent in the given binary code, which are not readily available from the syntactic analysis phase (S1). In particular, this step involves recovering the control-flow graphs (CFGs) and call graphs (CGs) in the binary code [37], [38]. Once the control-structural information is obtained, one can use any attribute of these control structures as a feature. We distinguish this step from semantic analysis (S3) because binary analysis frameworks typically provide CFGs and CGs for free; the analysts do not have to write a complex semantic analyzer.

(S3) Semantic Analysis. Using the control-structural information obtained from S2, one can perform traditional program analyses, such as data-flow analysis and symbolic analysis, on the binary to figure out the underlying semantics. In this step, one can generate features that represent sophisticated program semantics, such as how register values flow into various program points. One can also enhance the features gathered from S1–S2 along with the semantic information.

(S4) Vectorization and Comparison. The final step is to vectorize all the information gathered from S1–S3 to compute the similarity between the binaries. This step essentially results in a similarity score between 0 and 1.

Fig. 1 depicts the four-step process. The first three steps determine the inputs to the comparison step (S4), which are often referred to as *features*. Some of the first three steps can be skipped depending on the underlying features being used. The actual comparison methodology in S4 can also vary depending on the BCSA technique. For example, one may compute the Jaccard distance [39] between feature sets, calculate the graph edit distance [40] between CFGs, or even leverage deep learning algorithms [41], [42]. However, *as the success of any comparison algorithm significantly depends on the chosen features, this paper focuses on features used in previous studies rather than the comparison methodologies.*

In this section, we first describe the features used in the previous papers and their underlying assumptions (Section 2.1). We then discuss the benchmarks used in those papers and point out their problems (Section 2.2). Lastly, we present several research questions identified during our study (Section 2.3).

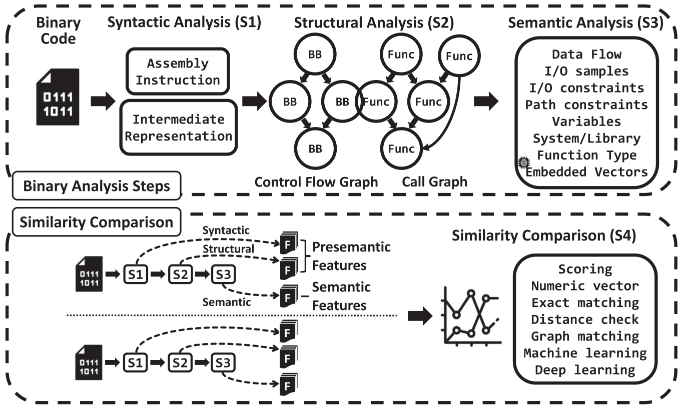


Fig. 1. Typical workflow of binary analysis (upper) and similarity comparison (lower) in binary code similarity analysis. Tools may skip some of the steps.

Scope. Our study focuses on 43 recent BCSA papers (from 2014 to 2020) that appeared in 27 top-tier venues of different computer science areas, such as computer security, software engineering, programming languages, and machine learning. There are, of course, plentiful research papers in this field, all of which are invaluable. Nevertheless, *our focus here is not to conduct a complete survey on them but to introduce a prominent trend and the underlying research questions in this field, as well as to answer those questions.*We particularly focus on features and datasets used in those studies, which lead us to four underexplored research questions that we will discuss in Section 2.3; our goal is to investigating these research questions by conducting a series of rigorous experiments. Because of the space limit, we excluded papers [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53] that were published before 2014 and those not regarding top-tier venues, or binary diffing tools [54], [55], [56] used in the industry. Additionally, we excluded papers that aimed to address a specific research problem such as malware detection, library function identification, or patch identification. Although our study focuses only on recent papers, we found that the features we studied in this paper are indeed general enough; they cover most of the features used in the older papers.

2.1 Features Used in Prior Works

We categorize features into two groups based on when they are generated during BCSA. Particularly, we refer to features obtained before and after the semantic analysis step (S3) as *presemantic features* and *semantic features*, respectively. Presemantic features can be derived from either S1 or S2, and semantic features can be derived from S3. We summarize both features used in the recent literature in Table 1.

2.1.1 Presemantic Features

Presemantic features denote direct or indirect outcomes of the syntactic (S1) and structural (S2) analyses. Therefore, we refer to any attribute of binary code, which can be derived without a semantic analysis, as a presemantic feature. We can further categorize presemantic features used in previous literature based on whether the feature represents a number or not. We refer to features representing a number as *numeric*

TABLE 1
Summary of the Features Used in Previous Studies

		2014						2015	2016							2017								
		TEDEM	Tracy	CoP	LoPD	BLEX	BinClone	Multi-k-MH	discovRE	Genius	Esh	BinGo	MockingBird	Kam1n0	BinDNN	BinSign	Xmatch	Gemini	GitZ	BinSim	BinSequence	IMF-sim	CACompare	ASE17
		[10]	[57]	[7]	[8]	[30]	[58]	[22]	[11]	[23]	[59]	[15]	[33]	[32]	[60]	[61]	[16]	[12]	[62]	[63]	[34]	[31]	[35]	[64]
Presemantic	BB-level Numbers	·	·	·	·	·	·	·	○	◑	·	·	·	·	·	·	·	◑	·	·	·	·	·	·
	CFG-level Numbers	·	·	·	·	·	○	·	○	◑	·	·	·	·	·	○	·	◑	·	·	○	·	·	·
	CG-level Numbers	·	·	·	·	·	·	·	○	·	·	○	·	·	·	○	·	·	·	·	·	·	·	·
	Raw Bytes	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Instructions	○	○	·	·	·	○	·	·	·	·	○	·	○	○	·	·	·	·	·	○	·	·	·
	Functions	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·
Semantic	Symbolic Constraints	·	·	○	○	·	·	·	·	·	○	·	·	·	·	·	○	·	·	○	·	·	·	·
	I/O Samples	·	·	·	○	·	·	○	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·
	Runtime Behavior	·	·	·	·	○	·	·	·	·	·	·	○	·	·	·	·	·	·	○	·	○	○	○
	Manual Annotation	·	·	·	·	·	·	·	·	·	·	○	·	·	·	○	·	·	·	○	·	·	·	·
	Program Slices, PDG	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·	·	·
	Recovered Variables	·	·	·	·	·	·	·	○	◑	·	·	·	·	·	○	·	◑	·	·	·	·	·	·
	Embedded Vector	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	○	·	·	·	·	·	·

		2018										2019						2020				
		BinArm	SANER18	BinGo-E	WSB	BinMatch	MASES18	Zeek	FirmUp	αDiff	VulSeeker	InnerEye	Asm2Vec	SAFE	BAR19i	BAR19ii	FuncNet	DeepBinDiff	ImOpt	ACCESS20	Patchecko	BinKit
		[17]	[65]	[18]	[66]	[67]	[25]	[68]	[14]	[19]	[13]	[24]	[20]	[21]	[26]	[29]	[69]	[27]	[70]	[53]	[28]	★
Presemantic	BB-level Numbers	○	·	·	·	·	·	·	·	·	◑	·	·	·	·	·	◑	·	·	·	◑	○
	CFG-level Numbers	○	○	○	·	·	·	·	·	·	◑	·	·	·	·	·	◑	·	·	·	◑	○
	CG-level Numbers	○	○	○	·	·	·	·	·	○	·	·	◑	·	·	·	·	·	·	·	◑	○
	Raw Bytes	·	·	·	·	·	◑	·	·	◑	·	·	·	·	·	·	·	·	·	○	·	·
	Instructions	·	·	○	·	·	·	·	·	·	·	◑	◑	◑	◑	◑	·	◑	◑	·	·	·
	Functions	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·
Semantic	Symbolic Constraints	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	I/O Samples	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Runtime Behavior	·	·	○	○	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	◑	·
	Manual Annotation	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Program Slices, PDG	·	·	·	·	·	·	○	○	·	◑	·	·	·	·	·	·	·	·	·	·	·
	Recovered Variables	○	○	○	·	·	·	·	·	·	·	·	·	·	·	·	◑	·	·	·	◑	·
	Embedded Vector	·	·	·	·	·	○	·	·	○	○	○	○	○	○	○	○	○	○	·	○	·

◑ *This mark denotes a feature that is not directly used for similarity comparison but is required for extracting other features used in post-processing.*

presemantic features, and others as *non-numeric presemantic features*. The first half of Table 1 summarizes them.

Numeric Presemantic Features. Counting the occurrences of a particular property of a program is common in BCSA as such numbers can be directly used as a numeric vector in the similarity comparison step (S4). We categorize numeric presemantic features into three groups based on the granularity of the information required for extracting them.

First, many researchers extract numeric features from each basic block of a target code snippet [11], [12], [13], [17], [23], [28], [28], [71]. One may measure the frequency of raw opcodes (mnemonics) [17], [71] or grouped instructions based on their functionalities (e.g., arithmetic, logical, or control transfer) [11], [28]. This numeric form can also be post-processed through machine learning [12], [13], [23], [28], as we further discuss in Section 2.1.2.

Similarly, numeric features can be extracted from a CFG as well. CFG-level numeric features can also reflect structural information that underlies a CFG. For example, a function can be encoded into a numeric vector, which consists of the number of nodes (i.e., basic blocks) and edges (i.e., control flow), as well as grouped instructions in its CFG [11], [28], [61]. One may extend such numeric vectors by adding extra features such as the number of successive nodes or the betweenness centrality of a CFG [12], [23], [28]. The concept of 3D-CFG [72], which places each node in a CFG onto a 3D space, can be utilized as well. Here, the distances among the centroids of two 3D-CFGs can represent their similarity score [18]. Other numeric features can be the graph energy, skewness, or cyclomatic complexity of a CFG [17], [28], [71]. Even loops in a CFG can be converted into numeric features by counting the number of loop headers and tails, as well as the number of forward and backward edges [65].

Finally, previous approaches utilize numeric features obtained from CGs. We refer to them as CG-level numeric features. Most of these approaches measure the number of callers and callees in a CG [11], [17], [19], [23], [28], [65], [71], [73]. When extracting these features, one can selectively apply an inter-procedural analysis using the ratio of the in-/out- degrees of the internal callees in the same binary and the external callees of imported libraries [15], [18], [20], [28]. This is similar to the coupling concept [74], which analyzes the inter-dependence between software modules. The extracted features can also be post-processed using machine learning [19].

Non-Numeric Presemantic Features. Program properties can also be directly used as a feature. The most straightforward approach involves directly comparing the raw bytes of binaries [6], [53], [75]. However, people tend to not consider this approach because byte-level matching is not as robust compared to simple code modifications. For example, anti-malware applications typically make use of manually written signatures using regular expressions to capture similar, but syntactically different malware instances [76]. Recent approaches have attempted to extract semantic meanings from raw binary code by utilizing a deep neural network (DNN) to build a feature vector representation [19], [25].

Another straightforward approach involves considering the opcodes and operands of assembly instructions or their intermediate representations [18], [77]. Researchers often normalize operands [32], [34], [57] because their actual values can significantly vary across different compiler options. Recent approaches [62], [70] have also applied re-optimization techniques [78] for the same reason. To compute a similarity score, one can measure the number of matched elements or the Jaccard distance [15] between matched groups, within a comparison unit such as a sliding window [58], basic block [34], or tracelet [57]. Here, a tracelet denotes a series of basic blocks. Although these approaches take different comparison units, one may adjust their results to compare two procedures, or to find the longest common subsequence [32], [34] within procedures. If one converts assembly instructions to a static single assignment (SSA) form, s/he can compute the tree edit distance between the SSA expression trees as a similarity score [10]. Recent approaches have proposed applying popular techniques in natural language processing (NLP) to represent an assembly instruction or a basic block as an embedded vector, reflecting their underlying semantics [20], [21], [24], [26], [27], [29].

Finally, some features can be directly extracted from functions. These features may include the names of imported functions, and the intersection of two inputs can show their similarity [19], [61]. Note that these features can collaborate with other features as well.

2.1.2 Semantic Features

We call the features that we can obtain from the semantic analysis phase (S3) *semantic features*. To obtain semantic features, a complex analysis, such as symbolic execution [7], [8], [15], [18], [63], dynamic evaluation of code snippets [8], [30], [31], [33], [35], [63], [64], [66], [67], or machine learning-based embedding [12], [13], [19], [20], [21], [23], [24], [25], [26], [27], [28], [29] is necessary. There are mainly seven distinct semantic features used in the previous literature, as listed in Table 1. It is common to use multiple semantic features together or combine them with presemantic features.

First, one straightforward method to represent the semantics of a given code snippet is to use symbolic constraints. The symbolic constraints could express the output variables or states of a basic block [7], a program slice [16], [59], [63], or a path [8], [79], [80]. Therefore, after extracting the symbolic constraints from a target comparison unit, one can compare them using an SMT solver.

Second, one may represent code semantics using I/O samples [8], [15], [18], [22]. The key intuition here is that two identical code snippets produce consistent I/O samples, and directly comparing them would be time-efficient. One can generate I/O samples by providing random inputs [8], [22] to a code snippet, or by applying an SMT solver to the symbolic constraints of the code snippet [15], [18]. One can also use inter-procedural analysis to precisely model I/O samples if the target code includes a function call [15], [18].

Third, the runtime behavior of a code snippet can directly express its semantics, as presented by traditional malware analysis [81]. By executing two target functions with the same execution environment, one can directly compare the executed instruction sequences [64] or visited CFG edges of the target functions [66]. For comparison, one may focus on specific behaviors observed during the execution [18], [28], [30], [31], [35], [67], [82]: the read/write values of stack and heap memory, return values from function calls, and invoked system/library function calls during the executions. To extract such features, one may adopt fuzzing [31], [83], or an emulation-based approach [67]. Moreover, one can further check the call names, parameters, or call sequences for system calls [18], [33], [35], [63], [67].

The next category is to manually annotate the high-level semantics of a program or function. One may categorize library functions by their high-level functionality, such as whether the function manipulates strings or whether it handles heap memory [15], [18], [61]. Annotating cryptographic functions in a target code snippet [84] is also helpful because its complex operations hinder analyzing the symbolic constraints or behavior of the code [63].

The fifth category is extracting features from a program slice [85], because they can represent its data-flow semantics in an abstract form. Specifically, one can slice a program into a set of strands [14], [62]. Here, a strand is a series of instructions within the same data flow, which can be obtained from backward slicing. Next, these strands can be canonicalized, normalized, or re-optimized for precise comparison [14], [62]. Additionally, one may hash strands for quick comparison [68] or extract symbolic constraints from the strands [59]. One may also extract features from a program dependence graph (PDG) [86], which is essentially a combination of a data-flow graph and CFG, to represent the convoluted semantics of the target code, including its structural information [13].

Recovered program variables can also be semantic features. For example, one can compare the similarity of string literals referenced in code snippets [11], [12], [17], [23], [28], [61], [65], [71]. One can also utilize the size of local variables, function parameters, or the return type of functions [11], [28], [61], [69]. One can further check registers or local variables that store the return values of functions [18].

Recently, several approaches have been utilizing embedding vectors, adopting various machine learning techniques. After building an attributed control-flow graph (ACFG) [23], which is a CFG containing numeric presemantic features in its basic blocks, one can apply spectral clustering [87] to group multiple ACFGs or popular encoding methods [88], [89], [90] to embed them into a vector [12]. The same technique can also be applied to PDGs [13]. Meanwhile, recent NLP techniques, such as Word2Vec [91] or convolutional neural network models [92], can be utilized for embedding raw bytes or assembly instructions into numeric vectors [19], [20], [21], [24], [25], [26], [27], [29]. For this embedding, one can also consider a higher-level granularity [20], [24] by applying other NLP techniques, such as sentence embedding [93] or paragraph embedding [94]. Note that one may apply machine learning to compare embedding vectors rather than generating them [60], [68], and Table 1 does *not* mark them to use embedded vectors.

2.1.3 Key Assumptions From Past Research

During our literature study, we found that most of the approaches highly rely on semantic features extracted in (S3), assuming that they should not change across compilers nor target architectures. However, none of them clearly justifies the necessity of such complex semantics-based analyses. They focus only on the end results without considering the precise reasoning behind their approaches.

This is indeed the key motivation for our research. Although most existing approaches focus on complex analyses, there may exist elementary features that we have overlooked. For example, there may exist effective presemantic features, which can beat semantic features regardless of target architectures and compilers. It can be the case that those known features have not been thoroughly evaluated on the right benchmark as there has been no comprehensive study on them.

Furthermore, existing research assumes the correctness of the underlying binary analysis framework, such as IDA Pro [95], which is indeed the most popular tool used, as shown in the rightmost column of Table 2. However, CFGs derived from those tools may be inherently wrong. They may miss some important basic blocks, for instance, which can directly affect the precision of BCSA features.

Indeed, both (S1) and (S2) are challenging research problems by themselves: there are abundant research efforts to improve the precision of both analyses. For example, disassembling binary code itself is an undecidable problem [96], and writing an efficient and accurate binary lifter is significantly challenging in practice [36], [97]. Identifying functions

TABLE 2
Summary of the Datasets Used in Previous Studies

		#Binaries*		Architecture								Optimization					Compiler†													Extra				Info.		
Year	Tool [Paper]	Packages	Firmware	x86	x64	arm	aarch64	mips	mips64	mipseb	mips64eb	O0	O1	O2	O3	Os	GCC 3	GCC 4	GCC 5	GCC 6	GCC 7	GCC 8	Clang 3	Clang 4	Clang 5	Clang 6	Clang 7	misc.	Total #	Noinline	PIE	LTO	Obfus.	Code	Dataset	IDA
2014	TEDEM [10]	14	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	Tracy [57]	(115)	·	△	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	○
	CoP [7]	(214)	·	△	·	·	·	·	·	·	·	○	○	○	○	○	·	1	·	·	·	·	·	·	·	·	·	1	2	·	·	·	○	·	·	○
	LoPD [8]	48	·	○	·	·	·	·	·	·	·	○	○	○	○	○	·	1	·	·	·	·	·	·	·	·	·	1	2	·	·	○	○	·	·	·
	BLEX [30]	1,140	·	·	○	·	·	·	·	·	·	○	○	○	○	·	·	1	·	·	·	·	1	·	·	·	·	1	3	·	·	·	·	·	·	○
	BinClone [58]	90	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	○
2015	Multi-k-MH [22]	60	6	○	·	○	·	○	·	·	·	○	○	○	○	·	·	2	·	·	·	·	1	·	·	·	·	·	3	·	·	·	·	·	·	○
2016	discovRE [11]	593	3	○	·	○	·	○	·	·	·	○	○	○	○	○	·	1	·	·	·	·	1	·	·	·	·	2	4	○	·	·	·	·	◑	○
	Genius [23]	(7,848)	8,128	○	·	○	·	○	·	·	·	○	○	○	○	·	·	2	·	·	·	·	1	·	·	·	·	·	3	·	·	·	·	·	·	○
	Esh [59]	(833)	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	3	·	·	·	·	2	·	·	·	·	2	7	·	·	·	·	○	○	○
	BinGo [15]	(5,143)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	·	3	·	·	·	·	1	·	·	·	·	1	5	·	·	·	·	·	·	○
	MockingBird [33]	80	·	○	·	○	·	○	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	·	·	·	○
	Kam1n0 [32]	96	·	○	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	○
	BinDNN [60]	(2,072)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	·	1	·	·	·	·	·	·	·	·	·	1	2	·	·	·	·	·	·	○
2017	BinSign [61]	(31)	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	2	2	·	·	·	○	·	·	○
	Xmatch [16]	72	1	○	·	·	·	○	·	·	·	·	·	·	·	·	·	2	·	·	·	·	1	·	·	·	·	·	3	○	·	·	·	·	·	○
	Gemini [12]	18,269	8,128	○	·	○	·	○	·	·	·	○	○	○	○	·	·	·	1	·	·	·	·	·	·	·	·	·	1	·	·	·	·	○	·	○
	GitZ [62]	44	·	·	○	·	○	·	·	·	·	○	○	○	○	○	·	3	·	·	·	·	2	1	·	·	·	2	8	·	·	·	·	·	·	·
	BinSim [63]	1,062	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	○
	BinSequence [34]	(1,718)	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	IMF-sim [31]	1,140	·	·	○	·	·	·	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	1	3	·	·	·	○	·	·	·
	CACompare [35]	72	·	○	·	○	·	○	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	·	·	·	○
	ASE17 [64]	55	·	○	○	·	·	·	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	○	·	·	·
2018	BinArm [17]	·	2,628	·	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	SANER18 [65]	7	·	○	○	·	·	·	·	·	·	·	·	·	·	·	·	1	1	1	·	·	1	·	·	·	·	1	5	·	·	·	·	·	·	○
	BinGo-E [18]	(5,145)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	·	3	·	·	·	·	1	·	·	·	·	1	5	·	·	·	·	·	·	○
	WSB [66]	(173)	·	△	·	·	·	·	·	·	·	·	○	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	○	·	·	·
	BinMatch [67]	(82)	·	○	·	·	·	·	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	○	·	·	○
	MASES18 [25]	47	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·
	Zeek [68]	(20,680)	·	·	○	·	○	·	·	·	·	○	○	○	○	○	·	3	·	·	·	·	4	1	·	·	·	2	10	·	·	·	·	·	·	·
	FirmUp [14]	·	2,000	△	·	△	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	αDiff [19]	(69,989)	2	○	○	○	·	·	·	·	·	○	○	○	○	·	·	2	1	·	·	·	2	·	·	·	·	·	5	·	·	·	·	◑	◑	○
	VulSeeker [13]	(10,512)	4,643	○	○	○	○	○	○	·	·	○	○	○	○	·	·	1	1	·	·	·	·	·	·	·	·	·	2	·	·	·	·	○	·	○
2019	InnerEye [24]	(844)	·	·	○	○	·	·	·	·	·	·	○	○	○	·	·	·	·	·	·	·	·	·	·	1	·	·	1	·	·	·	·	◑	◑	·
	Asm2Vec [20]	68	·	·	○	·	·	·	·	·	·	○	○	○	○	·	·	1	1	·	·	·	2	·	·	·	·	2	6	·	·	·	○	○	·	○
	SAFE [21]	(5,001)	·	·	○	○	·	·	·	·	·	○	○	○	○	·	1	3	1	1	1	·	2	1	1	1	·	·	12	·	·	·	·	○	◑	○
	BAR19i [26]	(804)	·	·	○	○	·	·	·	·	·	·	○	○	○	·	·	·	·	·	·	·	·	·	·	1	·	·	1	·	·	·	·	○	·	·
	BAR19ii [29]	(11,244)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	1	3	1	·	·	·	2	1	1	·	·	2	11	·	·	·	·	·	·	○
	FuncNet [69]	(180)	·	○	·	○	·	○	·	·	·	○	○	○	○	○	·	·	·	1	·	·	·	·	·	·	·	·	1	·	·	·	·	·	·	○
2020	DeepBinDiff [27]	(2,206)	·	·	△	·	·	·	·	·	·	○	○	○	○	·	·	1	·	·	·	·	·	·	·	·	·	·	1	·	·	·	·	○	○	○
	ImOpt [70]	18	·	·	○	·	·	·	·	·	·	○	·	○	○	·	·	·	1	·	·	·	·	·	·	·	·	·	1	·	·	·	○	·	·	·
	ACCESS20 [53]	12,000	·	△	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Patchecko [28]	2,108	2	○	○	○	○	·	·	·	·	○	○	○	○	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	BinKit ★	243,128	·	○	○	○	○	○	○	○	○	○	○	○	○	○	·	1	1	1	1	1	·	1	1	1	1	·	9	○	○	○	○	○	○	○

[∗] We only mark items that are stated explicitly in the paper. Due to the lack of details about firmware images, we were not able to mark optimization options or compilers used to create them. For papers that do not explicitly state the number of binaries in their dataset, we estimated the number and marked it with parentheses.
† This table focuses on two major compilers: GCC and Clang, as other compilers only support a limited number of architectures.
△ *We infer the target architectures of the dataset as they are not stated explicitly in the paper.*
◑ *This indicates that only a portion of the code and dataset is available. For example, discovRE [11] makes available only their firmware images, and αDiff [19] opens transformed function images but not the actual dataset.*

from binaries [3], [4], [96], [98], [99], [100], [101] and recovering control-flow edges [102] for indirect branches are still active research fields. All these observations lead us to research questions in Section 2.3.

2.2 Benchmarks Used in Prior Works

It is imperative to use the right benchmark to evaluate a BCSA technique. Therefore, we studied the benchmarks used in the past literature, as shown in Table 2. However, during the study, we found that it is radically difficult to properly evaluate a new BCSA technique using the previous benchmarks.

First, we were not able to find a single pair of papers that use the same benchmark. Some of them share packages such as GNU `coreutils` [15], [30], [31], but the exact binaries, versions, and compiler options are not the same. Although there is no known standard for evaluating BCSA, it is surprising to observe that none of the papers use the same dataset. We believe this is partly because of the difficulty in preparing the same benchmark. For example, even if we can download the same version of the source code used in a paper, it is extraordinarily difficult to cross-compile the program for various target architectures with varying compiler options; it requires significant effort to set up the environment. However, *only two out of 43 papers we studied fully open their dataset*. Even in that case, it is hard to rebuild or extend the benchmark because of the absence of a public compilation script for the benchmark.

Second, the number of binaries used in each paper is limited and may not be enough for analytics. The *#Binaries* column of Table 2 summarizes the number of program binaries obtained from two different sources: application packages and firmware images. Since a single package can contain multiple binaries, we manually extracted the packages used in each paper and counted the number of binaries in each package. We counted only the binaries after a successful compilation, such that the object files that were generated during the compilation process were not counted. If a paper does not explicitly mention package versions, we used the most recent package versions at the time of writing and marked them with parentheses. Note that only 6 out of 43 papers have more than 10,000 binaries, and none reaches 100,000 binaries. Firmware may include numerous binaries, but it cannot be directly used for BCSA because one cannot generate the ground truth without having the source code.

Finally, previous benchmarks only cover a few compilers, compiler options, and target architectures. Some papers do not even describe their tested compiler options or package versions. The *Compiler* column of the table presents the number of minor versions used for each major version of the compilers. Notably, all the benchmarks except one consider less than five different major compiler versions. The *Extra* column of the table shows the use of extra compiler options for each benchmark. Only a few consider function inlining and Link-Time Optimization (LTO). None of them deal with the Position Independent Executable (PIE) option, although, currently, it is widely used [103].

All these observations lead us to the research questions outlined in the next subsection (Section 2.3) and eventually motivate us to create our own benchmark that we call BinKit, which is shown in the last row of Table 2.

2.3 Research Problems and Questions

We now summarize several key problems observed from the previous literature and introduce research questions derived from these problems. First, none of the papers uses the same benchmark for their evaluation, and the way they evaluate their techniques significantly differs. Second, only a few of the studies release their source code and data, which makes it radically difficult to reproduce or improve upon existing works. Furthermore, most papers use manually chosen ground truth data for their evaluation, which are easily error-prone. Finally, current state-of-the-art approaches in BCSA focus on extracting semantic features with complex analysis techniques (from Sections 2.1.1 and 2.1.2). These observations naturally lead us to the below research questions. Note that some of the questions are indeed open-ended, and we only address them in part.

RQ1. How should we establish a large-scale benchmark and ground truth data?

One may build benchmarks by manually compiling application source code. However, there are so many different compiler versions, optimization levels, and options to consider when building binaries. Therefore, it is desirable to automate this process to build a large-scale benchmark for BCSA. It should be noted that many of the existing studies have also attempted to build ground truth from source code. However, the number of binaries and compiler options used in those studies is limited and is not enough for data-driven research. Furthermore, those studies release neither their source code nor dataset (Section 2.2). On the contrary, we present a script that can automatically build large-scale ground truth data from a given set of source packages with clear descriptions (Section 3).

RQ2. Is the effectiveness of presemantic features limited to the target architectures and compiler options used?

We note that most previous studies assume that presemantic features are significantly less effective than semantic features, as they can largely vary depending on the underlying architectures and compiler optimizations used. For example, compilers may perform target-specific optimization techniques for a specific architecture. Indeed, 36 out of the 43 papers ($\approx 84\%$) we studied focus on new semantic features in their analysis, as shown in Table 1. To determine whether this assumption is valid, we investigate it through a series of rigorous experimental studies. Although byte-level information significantly varies depending on the target and the optimization techniques, we found that some presemantic features, such as structural information obtained from CFGs, are broadly similar across different binaries of the same program. Additionally, we demonstrated that utilizing such presemantic features without a complex semantic analysis can achieve an accuracy that is comparable to that of a recent deep learning-based approach with a semantic analysis (Section 5).

RQ3. Can debugging information help BCSA achieve a high accuracy rate?

We are not aware of any quantitative study on how much debugging information affects the accuracy of BCSA. Most prior works simply assume that debugging information is not available, but how much does it help? How would decompilation techniques affect the accuracy of BCSA? To answer this question, we extracted a list of function types from our benchmark and used them to perform BCSA on our dataset. Surprisingly, we were able to achieve a higher accuracy rate than any other existing works on BCSA without using any sophisticated method (Section 6).

RQ4. Can we benefit from analyzing failure cases of BCSA?

Most existing works do not analyze their failure cases as they rely on uninterpretable machine learning techniques. However, our goal is to use a simple and interpretable model to learn from failure and gain insights for future research. Therefore, we manually examined failure cases using our interpretable method and observed three common causes for failure, which have been mostly overlooked by the previous literature. First, COTS binary analysis tools indeed return false results. Second, different compiler back-ends for the same architecture can be substantially different from each other. Third, there are architecture-specific code snippets for the same function. We believe that all these observations help in setting directions for future studies (Section 7).

Analysis Scope. In this paper, we focus on function-level similarity analyses because functions are a fundamental unit of binary analysis, and function-level BCSA is widely used in previous literature [11], [16], [19], [21], [24], [30], [31], [32]. We believe one can easily extend our work to support whole-binary-level similarity analyses as in the previous papers [7], [8].

3 Establishing Large-Scale Benchmark and Ground Truth for BCSA (RQ1)

Building a large-scale benchmark for BCSA and establishing its ground truth is challenging. One potential approach for generating the ground truth data is to manually identify similar functions from existing binaries or firmware images [10],

TABLE 3
Summary of BinKit

Dataset Name	# of Pkgs	# of Archs	# of Optis	# of Comps	# of Binaries	# of Orig. Functions	# of Final Functions*
Normal	51	8	4	9	67,680	34,355,824	8,708,459
SizeOpt	51	8	1†	9	16,920	8,350,442	2,060,625
Pie	46†	8	4	9	36,000	23,090,676	7,766,235
NoInline	51	8	4	9	67,680	38,617,186	10,291,001
Lto	29†	8	4	9	24,768	12,279,982	3,375,308
Obfuscation	51	8	4	4‡	30,080	15,809,489	4,054,694
Total	51	1,352 options			243,128	132,503,599	36,256,322

[∗] The target functions are selected in the manner described in Section 3.2.
† The number of packages and compiler options varies because some packages can be compiled only with a specific set of compile options.
‡ We count each of the four obfuscation options as a distinct compiler (Section 3.1).

[57], [59]. However, this requires domain expertise and is often error-prone and time-consuming.

Another approach for obtaining the ground truth is to compile binaries from existing source code with varying compiler options and target architectures [13], [15], [16], [23]. If we compile multiple binaries (with different compiler options) from the same source code, one can determine which function corresponds to which source lines. Unfortunately, most existing approaches do not open their benchmarks nor the compilation scripts used to produce them (Table 2).

Therefore, we present BinKit, which is a comprehensive benchmark for BCSA, along with automated compilation scripts that help reproduce and extend it for various research purposes. The rest of this section details BinKit and discusses how we establish the ground truth (RQ1).

3.1 BinKit: Large-Scale BCSA Benchmark

BinKit is a comprehensive BCSA benchmark that comprises 243,128 binaries compiled from 51 packages of source code with 1,352 distinct combinations of compilers, compilation options, and target architectures. Therefore, BinKit covers most of the benchmarks used in existing approaches, as shown in Table 2. BinKit includes binaries compiled for 8 different architectures. For example, we use both little- and big-endian binaries for MIPS to investigate the effect of endianness. It uses 9 different versions of compilers: GCC v{4.9.4, 5.5.0, 6.4.0, 7.3.0, 8.2.0} and Clang v{4.0, 5.0, 6.0, 7.0}. We also consider 5 optimization levels from `O0` to `O3` as well as `Os`, which is the code size optimization. Finally, we take PIE, LTO, and obfuscation options into account, which are less explored in BCSA.

We select GNU software packages [104] as our compilation target because of their popularity and accessibility: they are real applications that are widely used on Linux systems, and their source code is publicly available. We successfully compiled 51 GNU packages for all our target architectures and compiler options.

To better support targeted comparisons, we divide BinKit into six datasets: Normal, SizeOpt, NoInline, Pie, Lto, and Obfuscation. The summary of each dataset is shown in Table 3. Each dataset contains binaries obtained by compiling the GNU packages with different combinations of compiler options and targets. There is *no* intersection among the datasets.

Normal includes binaries compiled for 8 different architectures with different compilers and optimization levels. We did not use other extra options such as PIE, LTO, and no-inline for this dataset.

SizeOpt is the same as Normal except that it uses only the `Os` optimization option instead of `O0`–`O3`.

Similarly, Pie, NoInline, Lto, and Obfuscation are no different from Normal except that they are generated by using an additional flag to enable PIE, to disable inline optimization, to enable LTO, and to enable compile-time obfuscation, respectively.

PIE makes memory references in binary relative to support ASLR. On some architectures, e.g., x86, compilers inject additional code snippets to achieve relative addressing. As a result, the compiled output can differ severely. Although PIE became the default on most Linux systems [103], it has not been well studied for BCSA. Note we were not able to compile all 51 packages with the PIE option enabled. Therefore, we have fewer binaries in Pie than Normal.

Function inlining embeds callee functions into the body of the caller. This can make presemantic features largely vary. Therefore, we investigate the effect of function inlining on BCSA by explicitly turning off the inline optimization with the `fno-inline` option.

LTO is an optimization technique that operates at link time. It removes unnecessary code blocks, thereby reducing the number of presemantic features. However, it has also been less studied in BCSA. We were only able to successfully compile 29 packages when the LTO option was enabled.

Finally, the Obfuscation dataset uses Obfuscator-LLVM [105] to obfuscate the target binaries. We chose Obfuscator-LLVM among various other tools previously used [105], [106], [107], [108], [109], [110] because it is the most commonly used [20], [31], [61], [67], [70], and we can directly compare the effect of obfuscation using the vanilla LLVM compiler. We use Obfuscator-LLVM's latest version with four obfuscation options: instruction substitution (SUB), bogus control flow (BCF), control flow flattening (FLA), and a combination of all the options. We regard each option as a distinct compiler, as shown in the *Comp* column of Table 3. One can obfuscate a single binary multiple times. However, we only applied it once. This is because obfuscating a binary multiple times could emit a significantly large binary, which becomes time-consuming for IDA Pro to preprocess. For example, when we obfuscate `a2ps` twice with all three options, the compiled binary reaches over 30 MB, which is 30 times larger than the normal one.

The number of packages and that of compiler options used in compiling each dataset differ because some packages can be compiled only with a specific set of compile options and targets. Some packages fail to compile because they have architecture-specific code, such as inline assemblies, or because they use compiler-specific grammars. For example, Clang does not support both the LTO option and the `Os` option to be turned on. There are also cases where packages have conflicting dependencies. We also excluded the ones that did not compile within 30 min because some packages require a considerable amount of time to compile. For instance, `smalltalk` took more than 10 h to compile with the obfuscation option enabled.

To summarize, BinKit contains 243,128 binaries and 36,256,322 functions in total, which is indeed many orders of magnitude larger than the other benchmarks that appear in the previous literature. The *Source* column of Table 2 shows the difference clearly. BinKit does not include firmware images because our goal is to automatically build a benchmark with clear ground truth. One may extend our benchmark with firmware images. However, it would take significant manual effort to identify their ground truth. For additional details regarding each package, please refer to Table 12 in the Appendix, available online.

Our benchmark and compilation scripts are available on GitHub. Our compilation environment is based on Crosstool-NG [111], GNU Autoconf [112], and Linux Parallels [113]. Through this environment, we compiled the entire datasets of BinKit in approximately 30 h on our server machine with 144 Intel Xeon E7-8867v4 cores.

3.2 Building Ground Truth

Next, we establish the ground truth for our dataset. We first define the criteria for determining the equivalence of two functions. In particular, we check whether two functions with the same name originated from the same source files and have the same line numbers. Additionally, we verify that both functions come from the same package and have the same name in their binaries to ensure their equivalence.

Based on these criteria, we constructed the ground truth by performing the following steps. First, we compiled all the binaries with debugging information using the `-g` option. We then leveraged IDA Pro [95] to identify functions in the compiled binaries. Next, we labeled each identified function with its name, package name, binary name, as well as the name of the corresponding source file and line numbers. To achieve this, we wrote a script that parses the debugging information from each binary.

Using this information, we then sanitize our dataset to avoid having incorrect or biased results. Among the identified functions, we selected only the ones in the `code (.text)` segments, as functions in other segments may not include valid binary code. For example, we disregarded functions in the Procedure Linkage Table (`.plt`) sections because these functions are wrappers to call external functions and do not include actual function bodies. In our dataset, we filtered out 40% of the identified functions in this step.

We also disregarded approximately 4% of the functions that are generated by the compiler, but not by the application developers. We can easily identify such compiler intrinsic functions by checking the corresponding source files and line numbers. For example, GCC utilizes intrinsic functions such as `__udivdi3` in `libgcc2.c` or `__aeabi_uldivmod` in `bpabi.S` to produce highly optimized code.

Additionally, we removed duplicate functions within the same project/package. Two different binaries often share the same source code, especially when they are in the same project/package. For example, the GNU `coreutils` package contains 105 different executables that share 80% of the functions in common. We removed duplicate functions within each package by checking the source file names and their line numbers. Moreover, compilers can also generate multiple copies of the same function within a single binary due to optimization. These functions share the same source code but have a difference in their binary forms. For example, some parts of the binary code are removed or reordered for optimization purposes. As these functions share a large portion of the code, considering all of them would produce a biased result. To avoid this, we selected only one copy for each of the functions in our experiments. This step filtered out approximately 54% of the remaining functions. The last column of Table 3 reports the final counting results, which is the number of unique functions.

By performing all the above steps, we can automatically build large-scale ground truth data. The total time spent building the ground truth of all our datasets was 13,300 seconds. By leveraging this ground truth data, we further investigate the remaining research questions (i.e., RQ2–RQ4) in the following sections. To encourage further research, we have released all our datasets and source code.

4 Building an Interpretable Model

Previous BCSA techniques focused on achieving a higher accuracy by leveraging recent advances in deep learning techniques [12], [13], [19], [25]. This often requires building a complicated model, which is not straightforward to understand and hinders researchers from reasoning about the BCSA results and further answering the fundamental questions regarding BCSA. Therefore, we design an *interpretable* model for BCSA to answer the research questions and implement TikNib, which is a BCSA tool that employs the model. This section illustrates how we obtain such a model and how we set up our experimental environment.

4.1 TikNib Overview

At a high level, TikNib leverages a set of presemantic features widely used in the previous literature to reassess the effectiveness of presemantic features (RQ2). It evaluates each feature in two input functions, based on our similarity scoring metric (Section 4.3), which directly measures the difference between each feature value. In other words, it captures how much each feature differs across different compile options.

Note TikNib is intentionally designed to be simple so that we can answer the research questions presented in Section 2.3. Despite the simplicity of our approach, TikNib still produces a high accuracy rate that is comparable to state-of-the-art tools (Section 5.2). We are *not* arguing here that TikNib is the best BCSA algorithm.

4.2 Features Used in TikNib

Recall from RQ2, one of our goals is to reconsider the capability of presemantic features. Therefore, we focus on choosing various presemantic features used in the previous BCSA literature instead of inventing novel ones.

However, creating a comprehensive feature set is not straightforward because of the following two reasons. First, there are numerous existing features that are similar to one another, as discussed in Section 2. Second, some features require domain-specific knowledge, which is *not* publicly available. For example, several existing papers [11], [12], [13], [17], [18], [23], [61], [65] categorize instructions into

TABLE 4
Summary of Numeric Presemantic Features Used in TikNib

Category	Features	Count
CFG	# of basic blocks, edges, loops, SCCs, and back edges # of all, arith, data transfer, cmp, and logic instrs. # of shift, bit-manipulating, float, misc instrs. # of arith + shift, and data transfer + misc instrs. # of all/unconditional/conditional control transfer instrs. Avg. # of edges per a basic block Avg./Sum of basic block, loop, and SCC sizes Avg. # of all, arith, data transfer, cmp, and logic instrs. Avg. # of shift, bit-manipulating, float, misc instrs. Avg. # of arith + shift, and data transfer + misc instrs. Avg. # of all/unconditional/conditional control transfer instrs.	41
CG	# of callers, callees, imported callees # of incoming/outgoing/imported calls	6
	Total	47

semantic groups. However, grouping instructions is largely a subjective task, and there is no known standard for it. Furthermore, most existing works do not make their grouping algorithms public.

We address these challenges by (1) manually extracting representative presemantic features and (2) open-sourcing our feature extraction implementation. Specifically, we focus on numeric presemantic features. Because these features are represented as numbers, the relationship among their values across different compile options can be easily observed.

Table 4 summarizes the selected features. Our feature set consists of CFG- and CG-level numeric features as they can effectively reveal structural changes in the target code. In particular, we utilize features related to basic blocks, CFG edges, natural loops, and strongly connected components (SCCs) from CFGs, by leveraging NetworkX [114]. We also categorize instructions into several semantic groups based on our careful judgment by referring to the reference manuals [115], [116], [117] and leveraging Capstone [118]'s internal grouping. Next, we count the number of instructions in each semantic group per each function (i.e., CFG). Additionally, we take six features from CGs. The number of callers and callees represents a unique number of outgoing and incoming edges from CGs, respectively.

To extract these features, we conducted the following steps. First, we pre-processed the binaries in BinKit with IDA Pro [95]. We then generated the ground truth of these binaries as we described in Section 3.2. For those functions of which we have the ground truth, we extracted the aforementioned features. Table 5 shows the time spent for each of these steps. The IDA pre-processing took most of the time as IDA performs various internal analyses. Meanwhile, the feature extraction took much less time as it merely operates on the precomputed results from the pre-processing step.

4.3 Scoring Metric

Our scoring metric is based on the computation of the relative difference [119] between feature values. Given two functions A and B, let us denote a value of feature f for each function as A_f and B_f, respectively. Recall that any feature in TikNib can be represented as a number. We can compute the relative difference δ between the two feature values as follows:

$$\delta(A_f, B_f) = \frac{|A_f - B_f|}{|\max(A_f, B_f)|}. \quad (1)$$

TABLE 5
Breakdown of the Feature Extracting Time for BinKit

Dataset Name	IDA Pre-processing (s)	Ground Truth Building (s)	Feature Extraction (s)	Avg. Feature[†] Extraction (ms)
Normal	14,968.42	3,380.01	661.81	0.08
SizeOpt	2,171.70	353.13	649.57	0.32
Pie	13,893.92	2,601.74	133.60	0.02
NoInline	14,780.06	3,883.88	579.82	0.06
Lto	5,263.97	1,314.94	392.48	0.12
Obfuscation	97,723.47	1,766.60	4,189.44	1.03

[†] The average time spent for extracting features from a function, which is computed by dividing the total time (the fourth column of this table) by the number of functions (the last column of Table 3).

Let us suppose we have N distinct features $(f_1, f_2, \ldots, f_N)$ in our feature set. We can then define our similarity score s between two functions A and B by taking the average of relative differences for all the features as follows:

$$s(A, B) = 1 - \frac{\left(\delta(A_{f_1}, B_{f_1}) + \cdots + \delta(A_{f_N}, B_{f_N})\right)}{N}. \quad (2)$$

Although each numeric feature can have a different range of values, TikNib can effectively handle them using relative differences by representing the difference of each feature with a value between 0 and 1. Therefore, the score s is always within the range of 0 to 1.

Furthermore, we can intuitively understand and interpret the BCSA results using our scoring metric. For example, suppose there are two functions A and B derived from the same source code with and without compiler option X, respectively. If the relative difference of the feature value f between the two functions is small, it implies that f is a robust feature against compiler option X.

In this paper, we focus only on simple relative differences, rather than exploring complex relationships among the features for interpretability. However, we believe that our approach could be a stepping-stone toward fabricating more improved interpretable models to understand such complex relationships.

4.4 Feature Selection

Based on our scoring metric, we perform lightweight preprocessing to select useful features for BCSA as some features may not help in making a distinction between functions. To measure the quality of a given feature set, we compute the area under the receiver operating characteristic (ROC) curve (i.e., the ROC AUC) of generated models.

Suppose we are given a dataset in BinKit, which is generated from source code containing N unique functions. In total, we have a maximum of $N \cdot M$ functions in our dataset, where M is the number of combinations of compiler options used to generate the dataset. The actual number of functions can be less than $N \cdot M$ due to function inlining. For each unique function λ, we randomly select two other functions with the following conditions. (1) A true positive (TP) function, λ^{TP}, is generated from the same source code as in λ, with different compiler options, and (2) a true negative (TN) function, λ^{TN}, is generated from source code that is different from the one used to generate λ, with the same compiler options as for λ^{TP}. We generate such pairs for each unique function, thereby acquiring around $2 \cdot N$ function pairs. We then compute the similarity scores for the functions in each pair and their AUC.

We note that the same methodology has been used in prior works [12], [13]. We chose the method as it allows us to efficiently analyze the tendency over a large-scale dataset. One may also consider top-k [12], [13], [14], [20] or precision@k [12], [20] as an evaluation metric, but this approach has too much computational overhead: $O((N \cdot M)^2)$ operations.

Unfortunately, there is no efficient algorithm for selecting an optimal feature subset to use; it is indeed a well-known NP-hard problem [120]. Therefore, we leverage a greedy feature selection algorithm [121]. Starting from an empty set $\mathbb{F}$, we determine whether we can add a feature to $\mathbb{F}$ to increase its AUC. For every possible feature, we make a union with $\mathbb{F}$ and compute the corresponding AUC. We then select one that maximizes the AUC and update $\mathbb{F}$ to include the selected feature. We repeat this process until the AUC does not increase further by adding a new feature. Although our approach does not guarantee finding an optimal solution, it still provides empirically meaningful results, as we describe in the following sections.

4.5 Experimental Setup

For all experiments in this study, we perform 10-fold cross-validation on each test. When we split a test dataset, we ensure functions that share the same source code (i.e., source file name and line number) are either in a training or testing set, but not in both. For each fold, during the learning phase, i.e., the feature selection phase, we select up to 200K functions from a training set and conduct feature selection, as training millions of functions would take a significant amount of time. Limiting the number of functions for training may degrade the final results. However, when we tested the number of functions from 100K to 1000K, the results remained almost consistent. In the validation phase, we test all the functions in the testing set without any sampling. Thus, after 10-fold validation, all the functions in the target dataset are tested at least once.

We ran all our experiments on a server equipped with four Intel Xeon E7-8867v4 2.40 GHz CPUs (total 144 cores), 896 GB DDR4 RAM, and 8 TB SSD. We set up Ubuntu 18.04.5 LTS with IDA Pro v6.95 [95] on the server. For feature selection and similarity comparison, we utilized Python scikit-learn [122], SciPy [123], and NumPy [124].

5 Presemantic Feature Analysis (RQ2)

We now present our experimental results using TikNib on the presemantic features (Section 4.2) to answer RQ2 (Section 2.3). With our comprehensive analysis of these features, we obtained several useful insights for future research. In this section, we discuss our findings and lessons learned.

5.1 Analysis Result

To analyze the impact of various compiler options and target architectures on BCSA, we conducted a total of 72 tests using TikNib. We conducted the tests on our benchmark, BinKit, with the ground truth that we built in Section 3. Table 6 describes the experimental results where each column corresponds to a test we performed. Note that we present only 26 out of 72 tests because of the space limit. Unless otherwise specified, all the tests were performed on the Normal dataset. As described in Section 4.4, we prepared 10-fold sets for each test. We divided the tests into seven groups according to their purposes, as shown in the top row of the table. For example, the *Arch* group contains a set of tests to evaluate each feature against varying target architectures.

For each test, we select function pairs for training and testing as described in Section 4.4. That is, for a function λ, we select its corresponding functions (i.e., λ^{TP} and λ^{TN}). Therefore, N functions produce $2 \cdot N$ functions pairs. The first row (①) of Table 6 shows the number of function pairs for each test. When selecting these pairs, we deliberately choose the target options based on the goal of each test. For instance, we test the influence of varying the target architecture from x86 to ARM (*x86 versus ARM* column of Table 6). For each function λ in the x86 binaries of our dataset, we select both λ^{TP} and λ^{TN} from the ARM binaries compiled with the same compiler option as in λ. In other words, we fix all the other options, except for the target architecture for choosing λ^{TP} and λ^{TN} so we can focus on our testing goal. The same rule applies to other columns. For the *Rand.* columns, we alter all the compiler options in the group randomly to generate function pairs.

The second row (②) of Table 6 presents the time spent for training and testing in each test, which excludes the time for loading the function data on the memory. The average time spent for a single function was less than 1 ms.

Each cell in the third row (③) of Table 6 represents the average of $\delta(\lambda_f, \lambda_f^{\mathrm{TN}}) - \delta(\lambda_f, \lambda_f^{\mathrm{TP}})$ for feature f, which we call the *TP-TN gap* of f. This TP-TN gap measures the similarity between λ^{TP} and λ, as well as the difference between λ^{TN} and λ, in terms of the target feature. Thus, when the gap of a feature is larger, its discriminative capability for BCSA is higher. As we conduct 10-fold validation for each test, we highlight the cells with gray when the corresponding feature is chosen in all ten trials. Such features show relatively higher TP-TN gaps than the others do in each test. We also present the average TP-TN gaps in the fourth row (④) of the table.

The average number of the selected features in each test is shown in the fifth row (⑤) of Table 6. A few presemantic features could achieve high AUCs and average precisions (APs), as shown in the sixth row (⑥) and seventh row (⑦) of the same table, respectively. We now summarize our observations as follows.

5.1.1 *Optimization is Largely Influential*

Many researchers have focused on designing a model for *cross-architecture* BCSA [11], [15], [18], [22], [33]. However, our experimental results show that architecture may not be the most critical factor for BCSA. Instead, optimization level was the most influential factor in terms of the relative difference between presemantic features. In particular, we measured the average TP-TN gap of all the presemantic features for each test (*Avg. of TP-TN Gap* row of the table) and found that the average gap of the `O0` versus `O3` test (0.41) is less than that of the x86 versus ARM test (0.46) and the x86 versus MIPS test (0.42). Furthermore, the optimization level

TABLE 6
In-Depth Analysis Results of Presemantic Features Obtained by Running TikNib on BinKit

Index	Description	Opt Level				Compiler				Arch						vs. SizeOpt†			vs. Extra†			vs. Obfus.†				Bad‡	
		Rand.	O0 vs. O3	O2 vs. O3	Rand.	GCC v4 vs. GCC v8	Clang v4 vs. Clang v7	GCC vs. Clang	Rand.	x86 vs. ARM	x86 vs. MIPS	ARM vs. MIPS	32 vs. 64	LE vs. BE	Rand.	O0 vs. Os	O1 vs. Os	O3 vs. Os	PIE	NoInline	LTO	BCF	FLA	SUB	All	Norm.	Norm. vs. Obfus.†
①	# of Train Pairs (10^6)	0.40	0.13	0.19	0.36	0.19	0.19	0.19	0.40	0.17	0.16	0.17	0.18	0.20	0.39	0.14	0.17	0.18	6.04↑	0.17	0.19	0.19	0.19	0.20	0.18	3.63↑	4.56↑
	# of Test Pairs (10^6)	1.58	0.26	0.33	1.43	0.16	0.17	0.75	1.57	0.17	0.16	0.17	0.71	0.40	1.55	0.29	0.34	0.32	0.24	1.09	1.37	0.17	0.17	0.18	0.17	0.40↑	0.51↑
②	Train Time (sec)*	60.0	24.6	25.3	77.3	28.9	24.6	35.7	76.9	29.0	28.8	28.0	19.0	22.4	48.2	20.8	24.4	15.6	10.5	12.6	42.0	25.0	28.8	44.1	17.6	5.9	5.4
	Test Time (sec)*	59.1	23.3	23.8	49.7	12.0	11.8	40.9	54.4	12.3	12.5	12.0	39.9	22.8	52.2	24.8	23.2	22.6	32.0	62.5	60.7	11.9	11.3	10.6	11.2	1.6	1.6
③ TP-TN Gap of Features	CFG # of edges per BB	0.42	0.37	0.45	0.42	0.45	0.45	0.41	0.44	0.43	0.41	0.41	0.44	0.46	0.43	0.40	0.43	0.44	0.38	0.48	0.47	0.31	0.37	0.46	0.29	0.38	0.22
	CFG # of edges	0.57	0.50	0.68	0.58	0.68	0.69	0.57	0.64	0.67	0.62	0.63	0.67	0.72	0.64	0.54	0.63	0.60	0.63	0.66	0.72	0.40	0.37	0.72	0.31	0.52	0.25
	CFG # of loops	0.45	0.46	0.49	0.45	0.50	0.47	0.46	0.48	0.50	0.49	0.50	0.50	0.51	0.49	0.48	0.49	0.45	0.51	0.45	0.51	0.46	0.36	0.50	0.29	0.44	0.22
	CFG # of inter loops	0.46	0.47	0.49	0.46	0.50	0.48	0.47	0.48	0.50	0.49	0.50	0.50	0.50	0.49	0.48	0.49	0.45	0.51	0.45	0.50	0.46	0.38	0.50	0.32	0.45	0.27
	CG # of callees	0.57	0.52	0.63	0.58	0.64	0.63	0.57	0.61	0.64	0.57	0.58	0.60	0.64	0.59	0.54	0.62	0.60	0.56	0.61	0.62	0.61	0.61	0.64	0.60	0.52	0.56
	CG # of callers	0.55	0.55	0.59	0.56	0.60	0.58	0.54	0.58	0.58	0.51	0.53	0.56	0.60	0.57	0.58	0.60	0.58	0.54	0.58	0.57	0.56	0.56	0.58	0.55	0.54	0.54
	CG # of imported callees	0.55	0.56	0.63	0.58	0.62	0.59	0.55	0.59	0.63	0.49	0.50	0.59	0.57	0.57	0.58	0.61	0.58	0.58	0.59	0.59	0.56	0.57	0.59	0.56	0.52	0.55
	CG # of imported calls	0.56	0.57	0.65	0.59	0.65	0.62	0.56	0.61	0.67	0.50	0.51	0.62	0.60	0.59	0.59	0.62	0.59	0.61	0.60	0.62	0.53	0.59	0.62	0.52	0.52	0.51
	CG # of incoming calls	0.56	0.55	0.61	0.58	0.63	0.60	0.55	0.59	0.61	0.52	0.54	0.59	0.63	0.59	0.60	0.62	0.59	0.57	0.61	0.60	0.54	0.58	0.61	0.53	0.54	0.50
	CG # of outgoing calls	0.59	0.53	0.67	0.60	0.68	0.67	0.58	0.64	0.69	0.60	0.60	0.63	0.69	0.63	0.56	0.65	0.62	0.60	0.64	0.66	0.57	0.64	0.68	0.55	0.54	0.53
	Inst Avg. # of arith+shift	0.35	0.35	0.55	0.43	0.52	0.52	0.36	0.47	0.39	0.25	0.26	0.40	0.53	0.36	0.36	0.49	0.49	0.43	0.55	0.53	0.35	0.35	0.50	0.30	0.27	0.24
	Inst Avg. # of compare	0.44	0.40	0.52	0.46	0.51	0.52	0.44	0.49	0.45	0.44	0.37	0.50	0.55	0.46	0.43	0.50	0.49	0.46	0.54	0.55	0.33	0.37	0.54	0.29	0.39	0.21
	Inst Avg. # of ctransfer	0.34	0.33	0.45	0.37	0.42	0.42	0.33	0.40	0.39	0.36	0.38	0.39	0.46	0.37	0.35	0.39	0.39	0.38	0.43	0.46	0.30	0.28	0.43	0.24	0.31	0.18
	Inst Avg. # of ctransfer+cond.	0.27	0.25	0.32	0.28	0.32	0.31	0.26	0.30	0.29	0.28	0.29	0.29	0.33	0.28	0.28	0.29	0.29	0.28	0.33	0.34	0.23	0.20	0.32	0.17	0.22	0.11
	Inst Avg. # of dtransfer	0.34	0.30	0.49	0.38	0.46	0.49	0.35	0.43	0.35	0.31	0.33	0.38	0.52	0.36	0.31	0.43	0.43	0.37	0.49	0.49	0.35	0.33	0.47	0.27	0.31	0.19
	Inst Avg. # of dtransfer+misc	0.32	0.28	0.48	0.36	0.44	0.47	0.34	0.41	0.35	0.28	0.30	0.37	0.49	0.34	0.29	0.42	0.42	0.35	0.48	0.48	0.34	0.32	0.45	0.26	0.29	0.18
	Inst Avg. # of float instrs.	0.25	0.30	0.34	0.28	0.25	0.34	0.28	0.26	0.28	0.31	0.29	0.31	0.40	0.26	0.28	0.31	0.31	0.29	0.28	0.33	0.25	0.31	0.35	0.25	0.44	0.20
	Inst Avg. # of total instrs.	0.30	0.27	0.42	0.34	0.40	0.42	0.32	0.38	0.33	0.28	0.28	0.34	0.45	0.32	0.28	0.38	0.38	0.33	0.42	0.43	0.31	0.29	0.40	0.24	0.28	0.17
	Inst # of arith	0.40	0.43	0.64	0.51	0.62	0.61	0.46	0.55	0.43	0.28	0.29	0.48	0.62	0.41	0.44	0.58	0.56	0.53	0.61	0.60	0.40	0.50	0.58	0.35	0.33	0.27
	Inst # of arith+shift	0.40	0.43	0.64	0.51	0.62	0.61	0.46	0.55	0.44	0.27	0.28	0.47	0.63	0.41	0.43	0.58	0.56	0.53	0.61	0.61	0.40	0.50	0.59	0.35	0.32	0.27
	Inst # of bit-manipulating	0.26	0.29	0.35	0.28	0.33	0.32	0.26	0.30	0.19	0.13	0.20	0.33	0.17	0.25	0.27	0.32	0.30	0.33	0.31	0.33	0.25	0.23	0.24	0.18	0.41	0.03
	Inst # of compare	0.56	0.53	0.67	0.61	0.69	0.69	0.60	0.65	0.50	0.60	0.44	0.65	0.72	0.60	0.59	0.64	0.60	0.64	0.65	0.71	0.39	0.40	0.70	0.30	0.50	0.22
	Inst # of ctransfer	0.50	0.45	0.61	0.52	0.60	0.62	0.49	0.56	0.60	0.54	0.57	0.58	0.65	0.56	0.45	0.55	0.54	0.57	0.58	0.63	0.39	0.38	0.64	0.33	0.41	0.26
	Inst # of cond. ctransfer	0.60	0.54	0.68	0.61	0.69	0.70	0.63	0.67	0.69	0.66	0.67	0.69	0.72	0.67	0.60	0.64	0.60	0.64	0.65	0.72	0.39	0.37	0.71	0.31	0.52	0.25
	Inst # of dtransfer	0.42	0.36	0.63	0.46	0.61	0.61	0.48	0.55	0.48	0.37	0.41	0.50	0.64	0.46	0.34	0.57	0.56	0.49	0.61	0.61	0.41	0.39	0.61	0.33	0.38	0.26
	Inst # of dtransfer+misc	0.41	0.35	0.63	0.45	0.60	0.61	0.48	0.55	0.50	0.35	0.37	0.50	0.64	0.45	0.33	0.56	0.55	0.48	0.61	0.62	0.40	0.39	0.61	0.33	0.39	0.25
	Inst # of float instrs.	0.25	0.30	0.34	0.28	0.25	0.35	0.28	0.27	0.28	0.31	0.29	0.31	0.40	0.26	0.28	0.31	0.32	0.29	0.28	0.33	0.30	0.33	0.35	0.26	0.44	0.20
	Inst # of misc	0.30	0.26	0.52	0.34	0.48	0.48	0.33	0.42	0.11	0.31	0.27	0.46	0.67	0.34	0.23	0.39	0.38	0.38	0.48	0.50	0.30	0.31	0.50	0.26	0.32	0.15
	Inst # of shift	0.36	0.38	0.45	0.38	0.45	0.40	0.36	0.41	0.30	0.46	0.47	0.42	0.55	0.38	0.34	0.39	0.38	0.42	0.39	0.47	0.41	0.40	0.44	0.38	0.48	0.49
	Inst # of total instrs.	0.43	0.38	0.62	0.47	0.60	0.61	0.50	0.56	0.54	0.37	0.38	0.52	0.63	0.47	0.35	0.56	0.54	0.50	0.59	0.61	0.39	0.39	0.59	0.32	0.42	0.25
④	Avg. TP-TN Gap	0.42	0.41	0.53	0.45	0.52	0.52	0.44	0.49	0.46	0.42	0.43	0.49	0.55	0.46	0.42	0.49	0.48	0.48	0.51	0.54	0.39	0.38	0.53	0.32	0.42	0.27
	Avg. TP-TN Gap of Grey	0.49	0.44	0.54	0.49	0.59	0.60	0.52	0.56	0.57	0.52	0.50	0.59	0.60	0.57	0.49	0.56	0.54	0.53	0.57	0.53	0.48	0.48	0.57	0.44	0.47	0.45
⑤	Avg. # of Selected Features	8.5	13.9	9.3	12.9	10.1	8.7	11.7	11.0	11.0	12.0	11.0	7.1	8.4	7.3	11.0	10.0	5.7	14.3	5.3	16.5	8.3	9.9	15.7	6.1	11.6	8.1
⑥	ROC AUC	0.95	0.93	0.99	0.96	1.00	1.00	0.97	0.98	1.00	0.98	0.98	0.99	1.00	0.98	0.96	0.99	0.97	0.97	0.98	1.00	0.98	0.98	1.00	0.95	0.93	0.91
	Std. of ROC AUC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
⑦	Average Precision (AP)	0.95	0.93	0.99	0.97	1.00	1.00	0.97	0.99	0.99	0.97	0.98	0.99	1.00	0.98	0.96	0.99	0.98	0.98	0.98	1.00	0.98	0.98	1.00	0.95	0.93	0.90
	Std. of AP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01

All values in the table are 10-fold cross validation averages. We color a cell gray if a feature was consistently selected (i.e., 10 times) during the 10-fold validation. Due to the space constraints, we only display features that have been selected at least once during the 10-fold validation.
† *We compare a function from the Normal to the corresponding function in each target dataset.*
‡ *We match functions whose compiler options are largely distant to test for bad cases. Please refer to Section 5.1.8 for additional information.*
↑ *These in the first rows (①) are divided by 10^4 instead of 10^6.*
∗ *The train and test times in the seconds rows (②) do not include the time for data loading.*

random test (*Rand.* column of the *Opt Level* group) shows the lowest AUC (0.96) compared to that of the architecture and compiler group (0.98). These results confirm that compilers can produce largely distinct binaries depending on the optimization techniques used; hence, the variation among the binaries due to the optimization is considerably greater than that due to the target architecture on our dataset.

5.1.2 Compiler Version Has a Small Impact

Approximately one-third of the previous benchmarks shown in Table 2 employ multiple versions of the same compiler. However, we found that even the major versions of the same compiler produce similar binaries. In other words, compiler versions do not heavily affect presemantic features. Although Table 6 does not include all the tests we performed because of the space constraints, it is apparent from the *Compiler* column that the two tests between two different versions of the same compiler, i.e., GCC v4 versus GCC v8 and Clang v4 versus Clang v7, have much higher TP-TN gaps (0.52) than other tests, and their AUCs are close to 1.0.

5.1.3 GCC and Clang Have Diverse Characteristics

Conversely, the GCC versus Clang test resulted in the lowest TP-TN gap (0.44) and AUC (0.97) among the tests in the *Compiler* group. This can be because each compiler employs a different back-end, thereby producing different binaries. Another potential problem is that the techniques inside each optimization level can vary depending on the compiler. We detail this in Section 7.2.

5.1.4 ARM Binaries are Closer to x86 Binaries Than MIPS

The tests in the *Arch* group measure the influence of target architectures with the Normal dataset. Overall, the target architecture did not have much of an effect on the accuracy rate. The AUCs were over 0.98 in all the cases. Surprisingly, the x86 versus ARM test had the highest TP-TN gap (0.46) and AUC (1.0), indicating that the presemantic features of the x86 and ARM binaries are similar to each other, despite being distinct architectures. The ARM versus MIPS test showed a lower TP-TN gap (0.43) and AUC (0.98) although both of them are RISC architectures. Additionally, the effect of the word size (i.e., bits) and endianness was relatively small. Nevertheless, we cannot rule out the possibility that our feature extraction for MIPS binaries is erroneous. We further discuss this issue in Section 7.1.

5.1.5 Closer Optimization Levels Show Similar Results

We also measured the effect of size optimization (`Os`) by matching function λ in the NORMAL dataset with a function (λ^{TP} and λ^{TN}) in the SIZEOPT dataset. Subsequently, the binaries compiled with the `Os` option were similar to the ones compiled with the `O1` and `O2` options. This is not surprising because `Os` enables most of the `O2` techniques in both GCC and Clang [125], [126]. Furthermore, we observe that the `O1` and `O2` options produce similar binaries, although this is not shown in Table 6 due to the space limit.

5.1.6 Extra Options Have Less Impact

To assess the influence of the PIE, no-inline, and LTO options, we compared functions in the NORMAL dataset with those in the PIE, NOINLINE, and LTO datasets, respectively. For the no-inline test, we limit the optimization level from `O1` to `O3` as function inlining is applied from `O1`. It was observed that the influence of such extra options is not significant. Binaries with and without the PIE option were similar to each other because it only changes the instructions to use relative addresses; hence, it does not affect our presemantic features. Function inlining also does not affect several features, such as the number of incoming calls, which results in a high AUC (0.97). LTO does not exhibit any notable effects either.

However, by analyzing each test case, we found that some options affect the AUC more than others. For example, in the no-inline test, the AUC largely decreases as the optimization level increases: `O1` (0.995), `O2` (0.981), and `O3` (0.967). This is because as more optimization techniques are applied, more functions are inlined and transformed in the NORMAL, while their corresponding functions in the NOINLINE are not inlined. On the other hand, in the LTO test, the AUC increases as the version of Clang increases: v4 (0.956), v5 (0.968), v6 (0.986), and v7 (0.986). In contrast, GCC shows stable AUCs (0.987–0.988) across all versions, and all the AUCs are higher than those of Clang. This result indicates that varying multiple options would significantly affect the success rate, which we describe below.

5.1.7 Obfuscator-LLVM Does not Affect CG Features

Many previous studies [20], [31], [61], [67], [70] chose Obfuscator-LLVM [105] for their obfuscation tests as it significantly varies the binary code [20]. However, applying all of its three obfuscation options shows an AUC of 0.95 on our dataset, which is relatively higher than that of the optimization level tests. Obfuscation severely decreases the average TP-TN gaps except for CG features. This is because Obfuscator-LLVM applies intra-procedural obfuscation. The `SUB` obfuscation substitutes arithmetic instructions while preserving the semantics; the `BCF` obfuscation notably affects CFG features by adding bogus control flows; the `FLA` obfuscation changes the predicates of control structures [127]. However, none of them conducts inter-procedural obfuscation, which modifies the function call relationship. Thus, we encourage future studies to use other obfuscators, such as Themida [128] or VMProtect [107], for evaluating their techniques against inter-procedural obfuscation.

5.1.8 Comparison Target Option Does Matter

Based on the experimental results thus far, we perform extra tests to understand the influence of comparing multiple compiler options by intentionally selecting λ^{TP} and λ^{TN} from binaries that could provide the lowest TP-TN gap. In this study, we present two of them because of the space limit. Specifically, for the first test, we selected functions from 32-bit ARM binaries compiled using GCC v4 with the `O0` option, and the corresponding λ^{TP} and λ^{TN} functions from 64-bit MIPS big-endian binaries compiled using Clang v7 with the `O3` option. For the second test, we changed the Clang compiler to the Obfuscator-LLVM with all three obfuscation options turned on. The *Bad* column of the table summarizes the results. The AUC in both cases was approximately 0.93 and 0.91, respectively. Their average TP-TN gaps were also significantly lower (0.42 and 0.27) than those in the other tests. This signifies the importance of choosing the comparison targets for evaluating BCSA techniques. Existing BCSA research compares functions for all possible targets in a dataset, as shown in the *Rand.* tests in this study. However, our results suggest that researchers should carefully choose evaluation targets to avoid overlooking the influence of bad cases.

5.2 Comparison Against State-of-the-Art Techniques

From our experiments in Section 5.1, we show that using only presemantic features with a simple linear model (i.e., TIKNIB) is enough to obtain high AUC values. Next, we compare TIKNIB with state-of-the-art techniques.

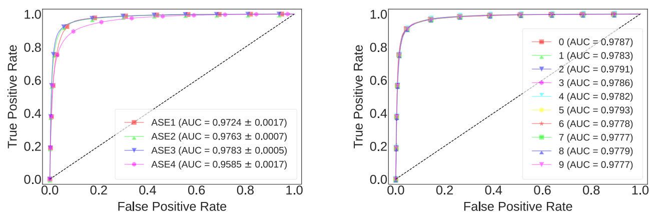
To accomplish this, we chose one of the latest approaches, VulSeeker [13], as our target because it utilizes both presemantic and semantic features in a numeric form by leveraging neural network-based post-processing. Thus, we can directly evaluate our simple model using numeric presemantic features. Note that *our goal is not to claim that our approach is better, but to demonstrate that the proper engineering of presemantic features can achieve results that are comparable to those of state-of-the-art techniques*.

For this experiment, we prepared the datasets of VulSeeker, along with the additional ones as listed in Table 7. We refer to these datasets as ASE1 through ASE4. ASE1 and ASE3 are the ones used in VulSeeker, and ASE2 and ASE4 are extra ones with more packages, target architectures, and compilers. Note that the number of packages, architectures, and compiler options increases as the index of the dataset increases. The optimization levels for all datasets are `O0`–`O3`. We intentionally omitted firmware images used in the original paper, as they do not provide solid ground truth. For each dataset, we established the ground truth in the same way described in Section 3.2. The time spent for IDA pre-processing, ground truth building, and feature extracting was 2197 s, 889 s, and 239 s, respectively. We then conducted experiments with the methodology explained in Section 4; note that the same methodology was used in the original paper.

TABLE 7
Summary of Datasets for Comparing TikNib to VulSeeker (i.e., ASE Datasets)

Name	Package	Architecture	Compiler (GCC)	# of Orig. Funcs	# of Final Funcs
ASE1	OpenSSL v1.0.1{f,u}	{x86,arm,mips}_32	v5.5.0	152K	126K
ASE2	OpenSSL v1.0.1{f,u} BusyBox v1.21 Coreutils v6.{5,7}	″	″	704K	183K
ASE3	″	{x86,arm,mips}_32, {x86,arm,mips}_64	v4.9.4, v5.5.0	2,777K	735K
ASE4	″	Same as Normal options		16,799K	4,467K

As the index of the dataset grows, the number of packages, architectures, and compiler options increases ASE1 and ASE3 are the datasets used in VulSeeker [13]. For all datasets, the optimization levels are `O0`*–*`O3`*.*



(a) ROC AUCs for all ASE datasets.

(b) ROC AUC of each fold for ASE3.

Fig. 2. Results obtained by running TikNib on ASE datasets.

Fig. 2 depicts the results. Fig. 2a shows that the AUCs of TikNib on ASE1 and ASE3 are 0.9724 and 0.9783, respectively. However, those of VulSeeker were 0.99 and 0.8849 as reported by the authors [13]. Fig. 2b illustrates that the AUC of each fold in ASE3 ranged from 0.9777 to 0.9793, which is higher than that of VulSeeker (0.8849). Therefore, TikNib was more robust than VulSeeker in terms of the size and compile options in the dataset. TikNib also exhibits stable results, even for ASE2 and ASE4.

From these results, we conclude that presemantic features combined with proper feature engineering can achieve results that are comparable to those of state-of-the-art BCSA techniques. Although our current focus is on comparing feature values, it is possible to extend our work to analyze the complex relationships among the features by utilizing advanced machine learning techniques [12], [13], [19], [20], [21], [23], [24], [25], [26], [27], [28], [29].

5.3 Analysis Case Study: Heartbleed (CVE-2014-0160)

To further assess the effectiveness of presemantic features, we apply TikNib to vulnerability discovery, which is a common practical application of BCSA [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. We investigate whether TikNib can effectively identify a vulnerable function across various compiler options and architectures.

We chose the `tls1_process_heartbeat` function in the `OpenSSL` package as our target function because it contains the infamous Heartbleed vulnerability (i.e., CVE-2014-0160), which has been widely used in prior studies for evaluation [11], [12], [20], [23]. We utilized two versions of `OpenSSL` in the ASE4 dataset shown in Table 7: v1.0.1f contains the vulnerable function, while v1.0.1u contains the patched version. As the dataset was compiled with 288 distinct combinations of compiler options and architectures, each function has 576 samples: 288 (the number of possible combinations) $\times$ 2 (the number of available `OpenSSL` versions) $\approx$ 576.

Notably, testing all possible combinations of options entails a significant computational overhead; it requires 288 (the number of options for our target function) $\times$ 287 (the number of options for a function in `OpenSSL`) $\times$ 2 (the number of `OpenSSL` versions) $\times$ 5K (the number of functions in `OpenSSL`) $\approx$ 826M operations. Therefore, we focused on architectures and compiler options that are widely used in software packages. Specifically, we chose three 64-bit architectures (aarch64, x86-64, and mips64el) and two levels of optimization (`O2`–`O3`). This setup reflects real-world scenarios, as many software packages use `O2`–`O3` by default: `coreutils` uses `O2`, while `OpenSSL` uses `O3`. Previous studies [20], [59] also used the same setup (`O2`–`O3`) except that they only tested x86 binaries. Additionally, we selected four compilers (Clang v4.0, Clang, v7.0, GCC v4.9.4, and GCC v8.2.0) to consider extreme cases. Consequently, there were 24 possible combinations of these architectures and compiler options.

We conducted a total of 552 tests on these 24 option combinations: 24 (the number of options for our target function) $\times$ 23 (the number of options for a function in `OpenSSL`). For each test, we simply computed the similarity scores for all function pairs using TikNib and checked the rank of the vulnerable function. To reflect real-world scenarios, we assumed in all tests that we were not aware of the precise optimization level, compiler type, or compiler version of the testing binary. On the other hand, we assumed that we could recognize the architecture of the testing binary as it is straightforward. Therefore, when we train TikNib, we chose a feature set that achieved the best performance across all possible combinations of optimization levels, compiler types, and compiler versions, while setting the source and target architectures fixed. For training, we used the Normal dataset (Table 3) as it does not include `OpenSSL`; thus, the training and testing datasets are completely distinct.

Table 8 summarizes the results, with each column corresponding to the tests for the specified options. We organized the results by the option group specified in each column after running all 522 tests. The first row of the table (*# of Option Pairs*) indicates the total number of option pairs, which is the same as that of true positive pairs. The remaining rows of the table show the averaged values obtained by the option pair tests. For example, the *All to All* column represents the averaged results of all possible combinations (24 $\times$ 23). The *ARM to MIPS* column, on the other hand, represents the averaged results of all combinations with the source and target architectures set to ARM and MIPS, respectively. That is, we queried the vulnerable functions compiled with ARM and searched for their true positives compiled with MIPS while varying the other options.

In the majority of the tests, TikNib successfully identified the vulnerable function with a rank close to 1.0 and a precision@1 close to 1.0, demonstrating its effectiveness in vulnerability discovery. Meanwhile, it performed marginally worse in the tests for MIPS. This result corroborates our observation in Section 5.1 that feature extraction for MIPS binaries can be erroneous. We further discuss this issue

TABLE 8
Real-World Vulnerability (Heartbleed, CVE-2014-0160) Analysis Result Using TikNib (Top-k and Precision@1)

Source option to Target option	All to All	ARM to ARM	ARM to MIPS	ARM to x86	MIPS to MIPS	MIPS to ARM	MIPS to x86	x86 to x86	x86 to ARM	x86 to MIPS	O2 to O3	O3 to O2	GCC to Clang	GCC v4 to GCC v8	GCC v8 to GCC v4	Clang v4 to Clang v7	Clang v7 to Clang v4
# of Option Pairs	552	56	64	64	56	64	64	56	64	64	144	144	144	36	36	36	36
Rank (tls, vuln)*	1.19	1.14	1.66	1	1	1.62	1	1	1.25	1	1.18	1.19	1	1.44	1.06	1	1
Precision@1 (tls, vuln)*	0.89	0.86	0.66	1	1	0.75	1	1	0.75	1	0.9	0.89	1	0.78	0.94	1	1
Rank (dtls, vuln)†	4.54	9.82	11.81	3.06	2	4.72	2	2.07	1.75	3.62	4.5	4.38	2.72	3.11	5.06	3.61	3.33
Rank (tls, patched)‡	29.16	12.12	57.69	3.56	3.82	51.62	43.94	4.29	6.38	70.59	27.5	28.96	27.68	32.89	40.89	20.22	22.67
Rank (dtls, patched)‡	76.47	46.95	145.75	7.25	8.21	128	128.94	9.57	11.94	181.03	73.04	75.41	87.31	66.28	87.33	68.44	78

We tested three 64-bit architectures (aarch64, x86-64, and mips64el) that are widely used in software packages.
All rank and precision@1 values are averaged; because each test involves multiple combinations of options (the first row), their results are averaged.
∗ *These are the results of the vulnerable* `tls1_process_heartbeat` *function in* `OpenSSL` *v1.0.1f.*
† *This is the result of the vulnerable* `dtls1_process_heartbeat`*function in* `OpenSSL` *v1.0.1f, which is similar to but distinct from the* `tls1_process_heartbeat` *function.*
‡ *These are the results of their patched versions in* `OpenSSL` *v1.0.1u.*

in Section 7.1. Additionally, the last three rows of Table 8 display the ranks of additional functions worth noting. The `dtls` represents the DTLS implementation of our target function (i.e., `dtls1_process_heartbeat`), which also contains the same vulnerability. Due to its similarity to our target function, it was ranked highly in all tests. The last two rows of the table present the ranks of the patched versions of these two functions in `OpenSSL` v1.0.1u. Notably, the patch of the vulnerability affects the presemantic features of these functions, particularly the number of control transfer and arithmetic instructions. Consequently, the patched functions had a low rank.

5.4 Analyzing Real-World Vulnerabilities on Firmware Images of IoT Devices

We further evaluate the efficacy of presemantic features by identifying vulnerable functions in real-world firmware images of IoT devices using TikNib. For the firmware images, we utilized the firmware dataset of FirmAE [129], which is one of the industry-leading large-scale firmware emulation frameworks. The dataset consists of 1,124 firmware images of wireless routers and IP cameras from the top eight vendors.

Particularly, we search for another infamous vulnerability (CVE-2015-1791) from `OpenSSL`, which has a race condition error in the `ssl3_get_new_session_ticket()` function. This vulnerability has also been extensively used in previous studies [12], [13], [23]. Since there exist numerous functions (≈52M) in the firmware images, identifying the vulnerable function among them is sufficient to evaluate the impact of presemantic features.

We evaluate our system, TikNib, against state-of-the-art techniques that support both ARM and MIPS architectures [12], [13]. Notably, these two architectures are prevalent in IoT devices [129]. However, while analyzing the repositories of these tools, we found that they did not include their complete source code nor datasets. As a result, we were unable to directly compare our system to theirs. Instead, we compared the results to the ones stated in the paper [13]. Specifically, we compiled the vulnerable version of `OpenSSL` (i.e., v1.0.1f) using a variety of compiler options and architectures, including six architectures (x86, ARM, and MIPS, each with 32 and 64 bits), two compilers (GCC v4.9.4 and v5.5.0), and two optimization levels (`O2`–`O3`). Here, we used two optimization levels (`O2`–`O3`) because many real-world software packages use them by default, as described in Section 5.3. Consequently, we obtained 24 samples of the vulnerable function. Notably, this dataset is essentially a subset of the ASE3 dataset, which is introduced in Table 7. Then, we queried each sample vulnerable function against all 52M functions in the 1,124 firmware images. This resulted in 24 similarity scores for each of the 52M functions. We then calculated the top-k result by averaging the similarity scores for each function. Finally, we manually counted the number of functions that were actually vulnerable in the top-100 results.

Table 9 summarizes the top-k results for the average similarity score for all 52M firmware functions. While our dataset is distinct from those used in the previous studies [12], [13], TikNib equipped with presemantic features achieved a level of performance comparable to that of the state-of-the-art tools. It should be noted that *our objective is not to assert that our approach is superior to the state-of-the-art tools, but rather to demonstrate the efficacy of appropriately utilizing presemantic features.* Additionally, our experimental results indicate that the real-world IoT firmware images (at least those that we tested) are highly likely to be compiled with `O2` or `O3`.

6 Benefit of Type Information (RQ3)

To assess the implication of debugging information on BCSA, we use type information as a case study on the presumption that they do not vary unless the source code is changed. Specifically, we extract three types of features per

TABLE 9
Top-k Results of Identifying CVE-2015-1791 for 52M Functions in 1,124 IoT Firmware Images Using TikNib

Top-k	Gemini†	VulSeeker†	TikNib (Ours)
1	1 (100%)	1 (100%)	1 (100%)
5	2 (40%)	3 (60%)	5 (100%)
10	4 (40%)	6 (60%)	10 (100%)
50	36 (72%)	41 (82%)	46 (92%)
100	75 (75%)	83 (83%)	82 (82%)

[†] Among 43 BCSA papers that we studied in Section 2, 10 released their source code, and two of these 10 support both ARM and MIPS architectures (Gemini [12] and VulSeeker [13]). However, we were not able to compare the results directly because these tools released neither their firmware datasets nor complete source code. Here, we present the results stated in the latest one [13]; note that their firmware dataset is different from the one that we used.

TABLE 10
In-Depth Analysis Results of Presemantic and Type Features Obtained by Running TikNib on BinKit

		Opt Level			Compiler				Arch						vs. SizeOpt†			vs. Extra†			vs. Obfus.†				Bad‡	
	Rand.	O0 vs. O3	O2 vs. O3	Rand.	GCC 4 vs. GCC 8	Clang 4 vs. Clang 7	GCC vs. Clang	Rand.	x86 vs. ARM	x86 vs. MIPS	ARM vs. MIPS	32 vs. 64	LE vs. BE	Rand.	O0 vs. Os	O1 vs. Os	O3 vs. Os	PIE	NoInline	LTO	BCF	FLA	SUB	All	Norm.	Norm. vs. Obfus.†
Avg. # of Selected Features	4.0	4.0	6.8	5.3	7.1	10.5	6.8	3.0	12.0	6.4	7.1	9.4	7.7	9.0	7.0	7.0	7.0	8.4	6.0	7.6	7.0	7.0	8.3	6.0	4.0	4.5
Avg. TP-TN Gap of Grey	0.53	0.52	0.56	0.53	0.58	0.59	0.53	0.56	0.54	0.54	0.54	0.56	0.56	0.55	0.54	0.57	0.57	0.56	0.56	0.55	0.50	0.52	0.58	0.50	0.55	0.55
ROC AUC	0.99	0.99	1.00	0.99	1.00	1.00	0.99	0.99	1.00	0.99	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.99	0.99	0.99
Average Precision (AP)	0.99	0.99	1.00	0.99	1.00	1.00	0.99	0.99	1.00	0.99	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.99	0.99	0.98

All values in the table are 10-fold cross validation averages.
† *We compare a function from the Normal to the corresponding function in each target dataset.*
‡ *We match functions whose compiler options are largely distant to test for bad cases. Please refer to Section 5.1.8 for additional information.*

function: the number of arguments, the types of arguments, and the return type of a function. Note that inferring the correct type information is challenging and is actively researched [130], [131]. In this context, we only consider basic types: `char`, `short`, `int`, `float`, `enum`, `struct`, `void`, and `void *`. To extract type information, we create a type map to handle custom types defined in each package by recursively following definitions using Ctags [132]. We then assign a unique prime number as an identifier to each type. To represent the argument types as a single number, we multiply their type identifiers.

To investigate the benefit of these type features, we conducted the same experiments described in Section 5, and Table 10 presents the results. Here, we explain the results by comparing them with Table 6, which we obtained without using the type features. The first row of Table 10 shows that the average number of selected features, including type features, is smaller than that of selected features (⑤) in Table 6. Note that all three type features were always selected in all tests. The second row in Table 10 shows that utilizing the type features could achieve a large TP-TN gap on average (over 0.50); the corresponding values in ④ of Table 6 are much smaller. Consequently, the AUC and AP with type features reached over 0.99 in all tests, as shown in the last two rows of Table 10. Additionally, it shows a similar result (i.e., an AUC close to 1.0) on the ASE datasets that we utilized for the state-of-the-art comparison (Section 5.2).

This result confirms that type information indeed benefits BCSA in terms of the success rate, although recovering such information is a difficult task. Therefore, we encourage further research on BCSA to take account of recovering debugging information, such as type recovery or inference, from binary code [130], [131], [133], [134], [135], [136].

7 Failure Case Inquiry (RQ4)

We carefully analyzed the failure cases in our experiments and found their causes. It was possible because our benchmark (i.e., BinKit) has the ground truth and our tool (i.e., TikNib) uses an interpretable model. We first checked the TP-TN gap of each feature for failure cases and further analyzed them using IDA Pro. We found that optimization largely affects the BCSA performance, as described in Section 5.1. In this section, we discuss other failure causes and summarize the lessons learned; however, many of these causes are closely related to optimization. We categorized the causes into three cases: (1) errors in binary analysis tools (Section 7.1), (2) differences in compiler back-ends (Section 7.2), and (3) architecture-specific code (Section 7.3).

7.1 Errors in Binary Analysis Tools

Most BCSA research heavily relies on COTS binary analysis tools such as IDA Pro [95]. However, we found that IDA Pro can yield false results. First, IDA Pro fails to analyze indirect branches, especially when handling MIPS binaries compiled with Clang using the position-independent code (PIC) option. The PIC option sets the compiler to generate machine code that can be placed in any address, and it is mainly used for compiling shared libraries or PIE binaries. Particularly, compilers use register-indirect branch instructions, such as `jalr`, to invoke functions in a position-independent manner. For example, when calling a function, GCC stores the base address of the Global Offset Table (GOT) in the `gp` register, and uses it to calculate the function addresses at runtime. In contrast, Clang uses the `s0` or `v0` register to store such base addresses. This subtle difference confuses IDA Pro and makes it fail to obtain the base address of the GOT, so that it cannot compute the target addresses of indirect branches.

Moreover, IDA Pro sometimes generates incomplete CFGs. When there is a *switch* statement, compilers often make a table that stores a list of jump target addresses. However, IDA Pro often failed to correctly identify the number of elements in the table, especially on the ARM architecture, where switch tables can be placed in a code segment. Sometimes, switch tables are located between basic blocks, and it is more difficult to distinguish them.

The problem worsens when handling MIPS binaries compiled for Clang with PIC, because switch tables are typically stored in a read-only data section, which can be referenced through a GOT. Therefore, if IDA Pro cannot fully analyze the base address of the GOT, it also fails to identify the jump targets of switch statements.

As we manually analyzed the errors, we may have missed some. Systematically finding such errors is a difficult task because the internals of many disassembly tools are not fully disclosed, and they differ significantly. One may extend the previous study [96] to further analyze the errors of disassembly tools and extracted features, and we leave this for future studies.

During the analysis, we found that IDA Pro also failed to fetch some function names if they had a prefix pre-defined in IDA Pro, such as `off_` or `sub_`. For example, it failed to fetch the name of the `off_to_chars` function in the `tar` package. We used IDA Pro v6.95 in our experiments, but we found that its latest version (v7.5) does not have this issue.

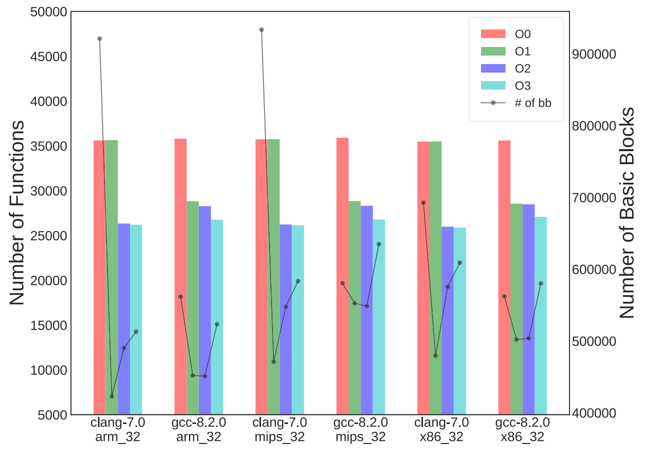


Fig. 3. The final number of functions and basic blocks in Normal (see Appendix for the detailed version), available in the online supplemental material.

7.2 Diversity of Compiler Back-Ends

From Section 5.1, the characteristics of binaries largely vary depending on the underlying compiler back-end. Our study reveals that GCC and Clang emit significantly different binaries from the same source code.

First, the number of basic blocks for the two compilers significantly differs. To observe how the number changes depending on different compiler options and target architectures, we counted the number for the Normal dataset. Fig. 3 illustrates the number of functions and basic blocks in the dataset for selected compiler options and architectures (see Appendix for details), available in the online supplemental material. As shown in the figure, the number of basic blocks in binaries compiled with Clang is significantly larger than that in binaries compiled with GCC for O0. We figured out that Clang inserts dummy basic blocks for O0 on ARM and MIPS; these dummy blocks have only one branch instruction to the next block. These dummy blocks are removed when the optimization level increases (O1) as optimization techniques in Clang merge such basic blocks into their predecessors.

In addition, the two compilers apply different internal techniques for the same optimization level, while they express the optimization level with the same terms (i.e., O0–O3 and Os). In particular, by analyzing the number of caller and callee functions, we discovered that GCC applies function inlining from O1, whereas Clang applies it from O2. Consequently, the number of functions for each compiler significantly differs (see the number of functions in O1 for Clang and that for GCC in Fig. 3).

Moreover, we discovered that two compilers internally leverage different function-level code for specific operations. For example, GCC has functions, such as `__umoddl3` in `libgcc2.c` or `__aeabi_dadd` in `ieee754-df.S`, to optimize certain arithmetic operations. Furthermore, on x86, GCC generates a special function, such as `__x86.get_pc_thunk.bx`, to load the current instruction pointer to a register, whereas Clang inlines this procedure inside the target function. These functions can largely affect the call-related features, such as the number of control transfer instructions or that of outgoing calls. Although we removed these compiler-specific functions so as not to include them in our experiments (Section 3.2), they may have been inlined in their caller functions in higher optimization levels (O2–O3). Considering such functions took approximately 4% of the identified functions by IDA Pro, they may have affected the resulting features.

Similarly, the two compilers also utilize different instruction-level codes. For example, in the case of move instructions for ARM, GCC uses conditional instructions, such as `MOVLE`, `MOVGT`, or `MOVNE`, unless the optimization level is zero (O0). In contrast, Clang utilizes regular move instructions along with branch instructions. This significantly affects the number of instructions as well as that of basic blocks in the resulting binaries. Consequently, in such special cases, the functions compiled using GCC have a relatively smaller number of basic blocks compared with those using Clang.

Finally, compilers sometimes generate multiple copies of the same function for optimization purposes. For example, they conduct inter-procedural scalar replacement of aggregates, removal of unused parameters, or optimization of cache/memory usage. Consequently, a compiled binary can have multiple functions that share the same source code but have different binary code. We found that GCC and Clang operate differently on this. Specifically, we discovered three techniques in GCC that produce function copies with special suffixes, such as `.part`, `.cold`, or `.isra`. For instance, for the `get_data` function of `readelf` in `binutils` (in O3), GCC yields three copies with the `.isra` suffix, while Clang does not produce any such functions. Similarly, for the `tree_eval` and `expr_eval` functions in `bool` (in O3), GCC produces two copies with the `.cold` suffix, but Clang does not. Although we selected only one such copy in our experiments to avoid biased results (Section 3.2), the other copies can still survive in their caller functions by inlining.

In summary, the diversities of compiler back-ends can largely affect the performance of BCSA, by making the resulting binaries divergent. Here, we have introduced the major issues we discovered. We encourage further studies to investigate the implications of detailed options at each optimization level across different compilers.

7.3 Architecture-Specific Code

When manually inspecting failures, we found that some packages have architecture-specific code snippets guarded with conditional macros such as `#if` and `#ifdef` directives. For example, various functions in `OpenSSL`, such as `mul_add` and `BN_UMULT_HIGH`, are written in architecture-specific inline assembly code to generate highly optimized binaries. This means that a function may correspond to two or more distinct source lines depending on the target architecture.

Therefore, instruction-level presemantic features can be significantly different across different architectures when the target programs have architecture-specific code snippets, and one should consider such code when designing cross-architecture BCSA techniques.

8 Discussion

Our study identifies several future research directions in BCSA. First, many BCSA papers have focused on building a general model that can result in stable outcomes with any compiler option. However, one could train a model targeting a specific set of compiler options, as shown in our experiment, to enhance their BCSA techniques. It is evident from our experiment's results that one can easily increase the success rate of their technique by inferring the compiler options used to compile the target binaries. There exists such an inference technique [137], and combining it with existing BCSA methods is a promising research direction.

Second, there are only a few studies on utilizing decompilation techniques for BCSA. However, our study reveals the importance of such techniques, and thus, invites further research on leveraging them for BCSA. One could also conduct a comprehensive analysis on the implication of semantic features along with decompilation techniques.

Additionally, we investigated fundamental presemantic features in this study. However, the effectiveness of semantic features is not well-studied yet in this field. Therefore, we encourage further research into investigating the effectiveness of semantic features along with other presemantic features that are not covered in the study. In particular, adopting NLP techniques would be another essential study as in many recent studies.

Our scope is limited to a function-level analysis (Section 4.1). However, one may extend the scope to handle other BCSA scenarios to compare binaries [20], [27], [54] or a series of instructions [32], [34], [57]. Additionally, one can extend our approach for various purposes, such as vulnerability discovery [11], [12], [20], [23], [28], [59], [138], malware detection [5], [6], [139], [140], [141], [142], [143], library function identification [71], [84], [144], [145], [146], [147], plagiarism/authorship detection [8], [82], [148], or patch identification [149], [150], [151]. However, extending our work to other BCSA tasks may not be directly applicable. This is because it requires additional domain knowledge to design an appropriate model that fits the purpose and careful consideration of the trade-offs. We believe that the reported insights in this study can help in this process.

Recall from Section 2, we did not intend to completely survey the existing techniques, but instead, we focused on systematizing the fundamental features used in previous literature. Furthermore, our goal was to investigate underexplored research questions in the field by conducting a series of rigorous experiments. For a complete survey, we refer readers to the recent surveys on BCSA [152], [153].

Finally, because our focus is on comparing binaries without source code, we intentionally exclude similarity comparison techniques that require source code. Nevertheless, it is noteworthy that there has been plentiful literature on comparing two source code snippets [75], [154], [155], [156], [157], [158], [159], [160], [161], [162] or comparing source snippets with binary snippets [163], [164], [165], [166].

9 Conclusion

We studied previous BCSA literature in terms of the features and benchmarks used. We discovered that none of the previous BCSA studies used the same benchmark for their evaluation, and that some of them required manually fabricating the ground truth for their benchmark. This observation inspired us to design BinKit, the first large-scale public benchmark for BCSA, along with a set of automated build scripts. Additionally, we developed a BCSA tool, TikNib, that employs an interpretable model. Using our benchmark and tool, we answered less-explored research questions regarding the syntactic and structural BCSA features. We discovered that several elementary features can be robust across different architectures, compiler types, compiler versions, and even intra-procedural obfuscation. Further, we proposed potential strategies for enhancing BCSA. We conclude by inviting further research on BCSA using our findings and benchmark.

Acknowledgments

The authors appreciate the anonymous reviewers for their thoughtful comments.

References

[1] S. P. Reiss, "Semantics-based code search," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 243–253.
[2] gpl-violations.org project prevails in court case on GPL violation by d-link, 2006. [Online]. Available: https://web.archive.org/web/20141007073104/http://gpl-violations.org/news/20060922-dlink-judgement_frankfurt.html
[3] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. USENIX Secur. Symp.*, 2015, pp. 611–626.
[4] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "ByteWeight: Learning to recognize functions in binary code," in *Proc. USENIX Secur. Symp.*, 2014, pp. 845–860.
[5] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 61–76.
[6] J. Jang, D. Brumley, and S. Venkataraman, "BitShred: Feature hashing malware for scalable triage and semantic analysis," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2011, pp. 309–320.
[7] 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 *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 389–400.
[8] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proc. IEEE Int. Symp. Softw. Rel. Eng.*, 2014, pp. 66–77.
[9] X. Meng, B. P. Miller, and K.-S. Jun, "Identifying multiple authors in a binary program," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2017, pp. 286–304.
[10] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 406–415.
[11] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016.
[12] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.
[13] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proc. ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 896–899.
[14] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 392–404.
[15] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-os binary search," in *Proc. Int. Symp. Found. Softw. Eng.*, 2016, pp. 678–689.
[16] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, 2017, pp. 346–359.

[17] P. Shirani *et al.*, "BinArm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2018, pp. 114–138.

[18] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-os binary code search with emulation," *IEEE Trans. Softw. Eng.*, vol. 45, no. 11, pp. 1125–1149, Nov. 2019.

[19] B. Liu *et al.*, "αdiff: Cross-version binary code similarity detection with DNN," in *Proc. ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 667–678.

[20] S. H. Ding, B. C. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 472–489.

[21] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-attentive function embeddings for binary similarity," in *Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2019, pp. 309–329.

[22] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 709–724.

[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2016, pp. 480–491.

[24] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019.

[25] N. Marastoni, R. Giacobazzi, and M. Dalla Preda, "A deep learning approach to program similarity," in *Proc. 1st Int. Workshop Mach. Learn. Softw. Eng. Symbiosis*, 2018, pp. 26–35.

[26] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," in *Proc. NDSS Workshop Binary Anal. Res.*, 2019.

[27] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning program-wide code representations for binary diffing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[28] P. Sun, L. Garcia, G. Salles-Loustau, and S. Zonouz, "Hybrid firmware analysis for known mobile and IoT security vulnerabilities," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2020, pp. 373–384.

[29] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proc. NDSS Workshop Binary Anal. Res.*, 2019.

[30] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proc. USENIX Secur. Symp.*, 2014, pp. 303–317.

[31] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 319–330.

[32] S. H. Ding, B. Fung, and P. Charland, "Kam1n0: MapReduce-based assembly clone search for reverse engineering," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 461–470.

[33] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *Proc. IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 57–67.

[34] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, accurate and scalable binary code reuse detection," in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, 2017, pp. 155–166.

[35] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proc. Int. Conf. Progr. Comprehension*, 2017, pp. 88–98.

[36] S. Kim *et al.*, "Testing intermediate representations for binary analysis," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 353–364.

[37] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proc. USENIX Secur. Symp.*, 2013, pp. 353–368.

[38] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[39] R. Real and J. M. Vargas, "The probabilistic basis of jaccard's index of similarity," *Systematic Biol.*, vol. 45, no. 3, pp. 380–385, 1996.

[40] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recognit. Lett.*, vol. 18, no. 8, pp. 689–694, 1997.

[41] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 1994, pp. 737–744.

[42] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, 2006.

[43] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proc. Int. Conf. Inf. Commun. Secur.*, 2008, pp. 238–255.

[44] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *SSTIC*, vol. 5, no. 1, 2005, Art. no. 3.

[45] H. Flake, "Structural comparison of executable objects," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2004, pp. 161–174.

[46] M. Bourquin, A. King, and E. Robbins, "BinSlayer: Accurate comparison of binary executables," in *Proc. 2nd ACM SIGPLAN Progr. Protection Reverse Eng. Workshop*, 2013, Art. no. 4.

[47] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *Proc. Int. Conf. Inf. Secur. Cryptol.*, 2012, pp. 92–109.

[48] W. Jin *et al.*, "Binary function clustering using semantic hashes," in *Proc. 11th Int. Conf. Mach. Learn. Appl.*, 2012, pp. 386–391.

[49] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic'juice'," in *Proc. 2nd ACM SIGPLAN Progr. Protection Reverse Eng. Workshop*, 2013, Art. no. 5.

[50] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digit. Investigation*, vol. 12, pp. S61–S71, 2015.

[51] S. Alrabaee, L. Wang, and M. Debbabi, "BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGS)," *Digit. Investigation*, vol. 18, pp. S11–S22, 2016.

[52] T. Kim, Y. R. Lee, B. Kang, and E. G. Im, "Binary executable file similarity calculation using function matching," *J. Supercomputing*, vol. 75, no. 2, pp. 607–622, 2019.

[53] H. Guo *et al.*, "A lightweight cross-version binary code similarity detection based on similarity and correlation coefficient features," *IEEE Access*, vol. 8, pp. 120 501–120 512, 2020.

[54] Bindiff. [Online]. Available: https://www.zynamics.com/bindiff.html

[55] Diaphora, a Free and Open Source program diffing tool. [Online]. Available: http://diaphora.re/

[56] J. W. Oh, "DarunGrim: A patch analysis and binary diffing too," 2015.

[57] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 349–360.

[58] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proc. Int. Conf. Softw. Secur. Rel.*, 2014, pp. 78–87.

[59] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2016, pp. 266–280.

[60] N. Lageman, E. D. Kilmer, R. J. Walls, and P. D. McDaniel, "BinDNN: Resilient function matching using deep learning," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, 2016, pp. 517–537.

[61] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, "BinSign: Fingerprinting binary functions to support automated analysis of code executables," in *Proc. IFIP Int. Conf. ICT Syst. Secur. Privacy Protection*, 2017, pp. 341–355.

[62] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 79–94.

[63] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proc. USENIX Secur. Symp.*, 2017, pp. 253–270.

[64] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 342–352.

[65] C. Karamitas and A. Kehagias, "Efficient features for function matching between binary executables," in *Proc. IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, 2018, pp. 335–345.

[66] B. Yuan, J. Wang, Z. Fang, and L. Qi, "A new software birthmark based on weight sequences of dynamic control flow graph for plagiarism detection," *Comput. J.*, vol. 61, pp. 1202–1215, 2018.

[67] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu, "BinMatch: A semantics-based hybrid approach on binary code clone analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 104–114.

[68] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proc. 13th Workshop Program. Lang. Anal. Secur.*, 2018, pp. 42–47.

[69] M. Luo, C. Yang, X. Gong, and L. Yu, "FuncNet: A euclidean embedding approach for lightweight cross-platform binary recognition," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, 2016, pp. 517–537.

[70] J. Jiang *et al.*, "Similarity of binaries across optimization levels and obfuscation," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2020, pp. 295–315.

[71] P. Shirani, L. Wang, and M. Debbabi, "BinShape: Scalable and robust binary library function identification using function shape," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2017, pp. 301–324.

[72] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 175–186.

[73] X. Hu, T.-C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2009, pp. 611–620.

[74] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. SE-7, no. 5, pp. 510–518, Sep. 1981.

[75] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire os distributions," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 48–62.

[76] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, "SplitScreen: Enabling efficient, distributed malware detection," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2010, pp. 377–390.

[77] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 329–338.

[78] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2013, pp. 305–316.

[79] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Trans. Rel.*, vol. 65, no. 4, pp. 1647–1664, Dec. 2016.

[80] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1157–1177, Dec. 2017.

[81] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proc. IEEE Symp. Secur. Privacy*, 1996, pp. 120–128.

[82] Z. Tian, Q. Wang, C. Gao, L. Chen, and D. Wu, "Plagiarism detection of multi-threaded programs via siamese neural networks," *IEEE Access*, vol. 8, pp. 160802–160814, 2020.

[83] V. J. M. Manès *et al.*, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.

[84] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2011, pp. 41–60.

[85] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1988, pp. 35–46.

[86] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[87] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Proc. Adv. Neural Inf. Process. Syst.*, 2002, pp. 849–856.

[88] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo, "Evaluating bag-of-visual-words representations in scene classification," in *Proc. Int. Workshop Multimedia Inf. Retrieval*, 2007, pp. 197–206.

[89] R. Arandjelovic and A. Zisserman, "All about VLAD," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2013, pp. 1578–1585.

[90] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2702–2711.

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

[92] Y. Kim, "Convolutional neural networks for sentence classification," 2014, *arXiv:1408.5882*.

[93] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[94] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.

[95] Hex-Rays, "IDA Pro," [Online]. Available: https://www.hex-rays.com/products/ida/

[96] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proc. USENIX Secur. Symp.*, 2016, pp. 583–600.

[97] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2R2: Building an efficient front-end for binary analysis," in *Proc. NDSS Workshop Binary Anal. Res.*, 2019.

[98] H. Kim, J. Lee, S. Kim, S. Jung, and S. K. Cha, "How'd security benefit reverse engineers? The implication of Intel CET on function identification," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2022, pp. 559–566.

[99] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2017, pp. 177–189.

[100] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 388–398.

[101] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in COTS binaries," in *Proc. Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2017, pp. 201–212.

[102] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proc. Int. Conf. Comput. Aided Verification*, 2008, pp. 423–427.

[103] SecurityTeam, "Pie," 2016. [Online]. Available: https://wiki.ubuntu.com/SecurityTeam/PIE

[104] GNU packages. [Online]. Available: https://ftp.gnu.org/gnu/

[105] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM–software protection for the masses," in *Proc. IEEE/ACM 1st Int. Workshop Soft. Protection*, 2015, pp. 3–9.

[106] M. Madou, L. Van Put, and K. De Bosschere, "LOCO: An interactive code (de) obfuscation tool," in *Proc. ACM SIGPLAN Symp. Partial Eval. Semantics-Based Progr. Manipulation*, 2006, pp. 140–144.

[107] VMProtect. [Online]. Available: http://vmpsoft.com

[108] Stunnix C/C++ Obfuscator. [Online]. Available: http://stunnix.com/prod/cxxo/

[109] Semantic Designs: Source Code Obfuscators. [Online]. Available: http://www.semdesigns.com/Products/Obfuscators/

[110] C. Collberg, "The tigress C diversifier/obfuscator," *Retrieved August*, vol. 14, 2015, Art. no. 2015.

[111] Crosstool-NG. [Online]. Available: https://github.com/crosstool-ng/crosstool-ng

[112] D. MacKenzie, B. Elliston, and A. Demaille, "Autoconf — Creating automatic configuration scripts," 1996. [Online]. Available: https://www.gnu.org/software/autoconf/manual/

[113] O. Tange, "GNU parallel - The command-line power tool," *;login: The USENIX Mag.*, vol. 36, no. 1, pp. 42–47, Feb. 2011. [Online]. Available: http://www.gnu.org/s/parallel

[114] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proc. Python Sci. Conf.*, 2008, pp. 11–15.

[115] Intel Corporation, "Intel 64 and ia-32 architectures software developer's manual," https://software.intel.com/en-us/articles/intel-sdm

[116] D. Seal, *ARM Architecture Reference Manual*, London, U.K.: Pearson Education, 2001.

[117] MIPS Technologies, Inc., "Mips32 architecture for programmers volume II: The mips32 instruction set," 2001.

[118] Capstone, "The ultimate disassembler," [Online]. Available: https://www.capstone-engine.org/

[119] Wikipedia, "Relative change and difference — Wikipedia, The free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Relative_change_and_differe%nce&oldid=872867886

[120] I. Guyon and A. Elisseeff, “An introduction to variable and feature selection,” vol. 3, no. Mar, pp. 1157–1182, 2003.
[121] R. Caruana and D. Freitag, “Greedy attribute selection,” in *Proc. 11th Int. Conf. Mach. Learn.*, 1994, pp. 28–36.
[122] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in python,” *J. Mach. Learn. Res.*, vol. 12, no. Oct, pp. 2825–2830, 2011.
[123] E. Jones *et al.*, “SciPy: Open source scientific tools for Python,” 2001. [Online]. Available: http://www.scipy.org/
[124] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, “The NumPy array: A structure for efficient numerical computation,” *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, 2011.
[125] Using the GNU compiler collection (GCC): Optimize options. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[126] Clang - The clang C, C++, and objective-C compiler. [Online]. Available: https://clang.llvm.org/docs/CommandGuide/clang.html
[127] T. László and Á. Kiss, “Obfuscating C++ programs via control flow flattening,” *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, pp. 3–19, 2009.
[128] Themida: Advanced windows software protection system. [Online]. Available: https://www.oreans.com/themida.php
[129] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, “FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis,” in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2020, pp. 733–745.
[130] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, “Neural nets can learn function type signatures from binaries,” in *Proc. USENIX Secur. Symp.*, 2017, pp. 99–116.
[131] V. van der Veen *et al.*, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 934–953.
[132] D. Hiebert, “Exuberant Ctags,” 1999.
[133] J. Lee, T. Avgerinos, and D. Brumley, “TIE: Principled reverse engineering of types in binary programs,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011.
[134] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, “Scalable variable and data type detection in a binary rewriter,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 51–60, 2013.
[135] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, “Debin: Predicting debug information in stripped binaries,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1667–1680.
[136] F. Artuso, G. A. Di Luna, L. Massarelli, and L. Querzoni, “In nomine function: Naming functions in stripped binaries with neural networks,” 2019, *arXiv:1912.07946*.
[137] N. Rosenblum, B. P. Miller, and X. Zhu, “Recovering the toolchain provenance of binary code,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 100–110.
[138] M. C. Tol, K. Yurtseven, B. Gulmezoglu, and B. Sunar, “FastSpec: Scalable generation and detection of spectre gadgets using neural embeddings,” 2020, *arXiv:2006.14147*.
[139] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “A quantitative study of accuracy in system call-based malware detection,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 122–132.
[140] D. Babić, D. Reynaud, and D. Song, “Malware analysis with tree automata inference,” in *Proc. Int. Conf. Comput. Aided Verification*, 2011, pp. 116–131.
[141] Y. Xiao *et al.*, “Matching similar functions in different versions of a malware,” in *Proc. IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 252–259.
[142] J. Ming, D. Xu, and D. Wu, “Memoized semantics-based binary diffing with application to malware lineage inference,” in *Proc. IFIP Int. Inf. Secur. Privacy Conf.*, 2015, pp. 416–430.
[143] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, “FOSSIL: A resilient and efficient system for identifying foss functions in malware binaries,” *ACM Trans. Privacy Secur.*, vol. 21, no. 2, pp. 1–34, 2018.
[144] J. Calvet, J. M. Fernandez, and J.-Y. Marion, “Aligot: Cryptographic function identification in obfuscated binary programs,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 169–182.
[145] D. Xu, J. Ming, and D. Wu, “Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping,” in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 921–937.
[146] J. Qiu, X. Su, and P. Ma, “Library functions identification in binary code by using graph isomorphism testings,” in *Proc. IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, 2015, pp. 261–270.
[147] L. Jia, A. Zhou, P. Jia, L. Liu, Y. Wang, and L. Liu, “A neural network-based approach for cryptographic function detection in malware,” *IEEE Access*, vol. 8, pp. 23 506–23 521, 2020.
[148] S. Alrabaee, M. Debbabi, and L. Wang, “CPA: Accurate cross-platform binary authorship characterization using LDA,” *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 3051–3066, Mar. 2020.
[149] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, “SPAIN: Security patch analysis for binaries towards understanding the pain and pills,” in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 462–472.
[150] Y. Hu, Y. Zhang, and D. Gu, “Automatically patching vulnerabilities of binary programs via code transfer from correct versions,” *IEEE Access*, vol. 7, pp. 28 170–28 184, 2019.
[151] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, “PatchScope: Memory object centric patch diffing,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2020, pp. 149–165.
[152] I. U. Haq and J. Caballero, “A survey of binary code similarity,” 2019, *arXiv:1909.11424*.
[153] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, “Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies,” *ACM Comput. Surv.*, vol. 54, no. 2, pp. 1–42, 2021.
[154] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.
[155] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local algorithms for document fingerprinting,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 76–85.
[156] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: A tool for finding copy-paste and related bugs in operating system code,” in *Proc. Symp. Oper. Syst. Des. Implementation*, 2004, pp. 289–302.
[157] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” in *Proc. Int. Conf. Softw. Eng.*, 2007, pp. 96–105.
[158] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, “Automatic feature learning for vulnerability prediction,” 2017, *arXiv:1708.02368*.
[159] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, “SymDiff: A language-agnostic semantic diff tool for imperative programs,” in *Proc. Int. Conf. Comput. Aided Verification*, 2012, pp. 712–717.
[160] S. Kim, S. Woo, H. Lee, and H. Oh, “VUDDY: A scalable approach for vulnerable code clone discovery,” in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 595–614.
[161] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 297–308.
[162] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “VulPecker: An automated vulnerability detection system based on code similarity analysis,” in *Proc. Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 201–213.
[163] D. Miyani, Z. Huang, and D. Lie, “BinPro: A tool for binary source code provenance,” 2017, *arXiv:1711.00830*.
[164] A. Rahimian, P. Charland, S. Preda, and M. Debbabi, “RESource: A framework for online matching of assembly with open source code,” in *Proc. Int. Symp. Found. Pract. Secur.*, 2012, pp. 211–226.
[165] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, “Finding software license violations through binary code clone detection,” in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 63–72.
[166] Y. Ji, L. Cui, and H. H. Huang, “BugGraph: Differentiating source-binary code similarity with graph triplet-loss network,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2021, pp. 702–715.

Dongkwan Kim received the PhD degree from the School of Electrical Engineering, Korea Advanced Institute of Science and Technology. He is a freelancer security researcher. His research interests include securing software, embedded & cyber-physical systems, and cellular infrastructures. He competed in various hacking contests, such as DEFCON, Codegate, and Whitehat Contest.

Eunsoo Kim received the PhD degree from the Graduate School of Information Security, KAIST. He is a security researcher with Samsung Research. His research interests include finding vulnerabilities in various software and embedded systems.

Sang Kil Cha received the PhD degree from the Electrical & Computer Engineering Department, Carnegie Mellon University. He is an Associate Professor of computer science with KAIST. His current research interests revolve mainly around software security, software engineering, and program analysis. He received an ACM distinguished Paper Award, in 2014. He is currently supervising GoN and KaisHack, which are, respectively, undergraduate and graduate hacking team with KAIST.

Sooel Son received the PhD degree from the Department of Computer Science, University of Texas at Austin. He is an Associate Professor of the School of Computing, KAIST. He is working on various topics regarding web security and privacy.

Yongdae Kim received the PhD degree from Computer Science Department, University of Southern California. He is a professor with the Department of Electrical Engineering, and an affiliate professor with the Graduate School of Information Security, KAIST. Between 2002 and 2012, he was a professor with the Department of Computer Science and Engineering, University of Minnesota - Twin Cities. Before coming to the US, he worked 6 years in ETRI for securing Korean cyber-infrastructure. He served as a KAIST chair professor between 2013 and 2016, and received NSF Career Award on storage security and McKnight Land-Grant Professorship Award from University of Minnesota, in 2005. His main research includes novel attacks and analysis methodologies for emerging technologies, such as 4G/5G cellular networks, drone/self-driving cars, and blockchain.