Binary Cryptographic Function Identification via Similarity Analysis with Path-Insensitive Emulation

YIKUN HU*, Shanghai Jiao Tong University, China and State Key Laboratory of Cryptology, China
YITUO HE, Shanghai Jiao Tong University, China
WENYU HE, Shanghai Jiao Tong University, China
HAORAN LI, Shanghai Jiao Tong University, China
YUBO ZHAO, Shanghai Jiao Tong University, China
SHUAI WANG, The Hong Kong University of Science and Technology, China
DAWU GU*, Shanghai Jiao Tong University, China and State Key Laboratory of Cryptology, China

It becomes an essential requirement to identify cryptographic functions in binaries due to their widespread application in modern software. The technology fundamentally supports numerous software security analyses, such as malware analysis, blockchain forensics, etc. Unfortunately, the existing methods still struggle to strike a balance between analysis accuracy, efficiency, and code coverage, which hampers their practical application.

In this paper, we propose BinCrypto, a method of emulation-based code similarity analysis on the interval domain, to identify cryptographic functions in binary files. It produces *accurate* results because it relies on the behavior-related code features collected during emulation. On the other hand, the emulation is performed in a path-insensitive manner, where the emulated values are all represented as intervals. As such, it is able to analyze every basic block only once, accomplishing the identification *efficiently*, and achieve *complete block coverage* simultaneously. We conduct the experiments with nine real-world cryptographic libraries. The results show that BinCrypto achieves the average accuracy of *83.2%*, nearly twice that of WheresCrypto, the state-of-the-art method. BinCrypto is also able to successfully complete the tasks, including statically-linked library analysis, cross-library analysis, obfuscated code analysis, and malware analysis, demonstrating its potential for practical applications.

CCS Concepts: • **Security and privacy** → **Software reverse engineering**.

Additional Key Words and Phrases: Static Analysis, Binary Analysis, Similarity Analysis, Cryptographic Function Identification

ACM Reference Format:
Yikun Hu, Yituo He, Wenyu He, Haoran Li, Yubo Zhao, Shuai Wang, and Dawu Gu. 2025. Binary Cryptographic Function Identification via Similarity Analysis with Path-Insensitive Emulation. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 81 (April 2025), 29 pages. https://doi.org/10.1145/3720415

*Corresponding authors.

Authors' Contact Information: Yikun Hu, yikunh@sjtu.edu.com, Shanghai Jiao Tong University, Shanghai, China and State Key Laboratory of Cryptology, Beijing, China; Yituo He, yituo_he@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Wenyu He, wenyu_he@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Haoran Li, haoranli@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Yubo Zhao, yubozhao@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Shuai Wang, shuaiw@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Dawu Gu, dwgu@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China and State Key Laboratory of Cryptology, Beijing, China.

ACM 2475-1421/2025/4-ART81
https://doi.org/10.1145/3720415

1 Introduction

Cryptography plays a vital role in modern software to protect confidentiality, integrity, and authenticity, such as in cryptocurrency wallets [Han et al. 2021; Houy et al. 2023], SSL/TLS protocol [Oppliger 2009], etc. Its misuse or misimplementation may cause devastating consequences (e.g., POODLE vulnerability [POODLE 2014]). It also could be abused in malicious software to threaten cybersecurity. Ransomware, for example, is a type of malware that blocks access to a victim's data by encryption for ransom [O'Gorman and McDonald 2012]. Therefore, to ensure software security, it is essential to identify the cryptographic implementations in binaries, which facilitates the downstream cryptography-related security analysis, e.g., cryptographic vulnerability detection, malware analysis, etc.

The existing methods generally focus on identifying standard cryptographic functions. Despite the considerable efforts in this field, they still struggle to strike a balance among various analysis metrics, including accuracy, efficiency, and code coverage. One typical category of those methods employs heuristic patterns to accomplish the identification. They have difficulty in producing accurate results, although they are fast and cover all the code. For example, FindCrypt2 [Guilfanov 2006] and Signsrch [Auriemma 2016] search the constant values of cryptographic algorithms. ReFormat [Wang et al. 2009] observes the ratio of bitwise operations for the identification. Due to the code transformation (e.g., compiler optimization), the resultant binaries would exhibit significant differences in syntax and structure, although they are compiled from the same codebase [Egele et al. 2014]. These methods cannot generally identify variant implementations of the same cryptographic algorithm accurately.

Recently, more sophisticated program analysis techniques have been adopted for the identification. However, they still suffer from the inherent limitations of static and dynamic analysis to solve the aforementioned problem. The static methods rely on the isomorphism of data flow graphs, which are unstable across code transformation, leading to relatively lower accuracy [Lestringant et al. 2015; Meijer et al. 2021]. While WheresCrypto [Meijer et al. 2021] is able to handle both standard and proprietary algorithms, when confronted with code transformation, it still underperforms in analyzing even standard cryptographic functions (§5.3). By contrast, the dynamic methods depend on input-output relations or loop structures captured from execution traces to achieve the goal, while they only cover limited code triggered by the input [Calvet et al. 2012; Gröbert et al. 2011; Li et al. 2012; Xu et al. 2017b]. Besides, since execution traces usually contain large amounts of data, dynamic methods tend to be time-consuming, which hinders their practical application.

In this paper, we propose BinCrypto, a method of emulation-based code similarity analysis with intervals, to identify cryptographic functions of known algorithms[1] in binaries. Different implementations of the same cryptographic algorithm have equivalent semantics. Therefore, given the same input, their output should be similar. BinCrypto emulates the execution of binary code with pre-defined input, performing similarity analysis based on the emulated behavior-related values. Then, it is able to handle variant binary code of the same algorithm. Additionally, the emulation is performed in a path-insensitive manner. The emulated values are merged as intervals at specific program points to avoid path explosion. Meanwhile, all basic blocks are covered, and each one is processed only once. In this manner, BinCrypto is practical for striking a balance between analysis accuracy, efficiency, and code coverage.

Specifically, BinCrypto involves the following main steps. Given the binary code, it first recovers the static information of binary functions via preprocessing. Then, it emulates each function with pre-defined input, collecting behavior-related values during the process. Finally, BinCrypto calculates the similarity score by comparing the values to those of the reference implementation.

[1] Standard algorithms or proprietary ones which have been analyzed or understood.

The function with the highest score is considered to implement the same cryptographic algorithm as the reference.

We evaluate BinCrypto with nine real-world binary cryptographic libraries on the x64 Linux platform. The experimental results show that BinCrypto is able to identify and distinguish variant cryptographic functions which are compiled with various compilers and optimization options. It attains the average accuracy of *83.2%*, nearly twice that of WheresCrypto, the state-of-the-art method of cryptographic function identification, with less processing time. It also surpasses five other representative methods regarding analysis accuracy by a large margin. Furthermore, we show the potential of BinCrypto for practical applications, including statically-linked library analysis, cross-library analysis, obfuscated code analysis, and malware analysis.

In summary, the paper makes the following contributions.

- We propose BinCrypto, a method based on code similarity analysis to identify cryptographic functions of known algorithms in binaries. It relies solely on behavior-related values as code features to accurately identify variant implementations of the same cryptographic algorithm.
- BinCrypto adopts path-insensitive emulation to extract binary code features, and the emulated variable values are represented as intervals. In this way, it is able to achieve complete block coverage efficiently.
- We have implemented a prototype of BinCrypto on the 64-bit Linux platform. It is evaluated with nine real-world cryptographic libraries. The results show that BinCrypto is much more accurate than the baselines, and spends less processing time than WheresCrypto, the state-of-the-art method. We also demonstrate its potential for practical application in statically-linked library analysis, cross-library analysis, and malware analysis. The extended version of the paper is provided at [Hu et al. 2025] which includes full details of this work.

2 Motivation and Overview

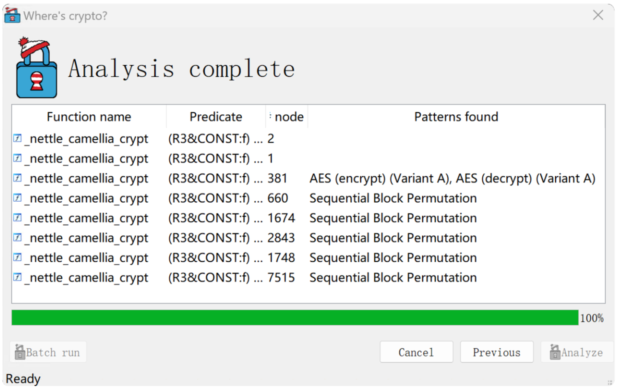
In this section, we first outline the limitations of existing techniques in binary cryptographic code identification and summarize the challenge of the problem, which motivate BinCrypto. Then, we explain the basic idea of BinCrypto and present an overview of the system.

2.1 Limitations of Existing Techniques

2.1.1 Heuristic-Based Techniques. One category of classic techniques adopts constants as designed in cryptographic algorithms to identify their implementations in binaries. They search for such patterns in the data and code sections of binary files. For example, Signsrch [Auriemma 2016] relies on S-box values to detect AES in NSS [Mozilla [n. d.]]. However, they become less effective in handling code variants in practice, which might hide those constants to prevent cryptanalysis attacks [Das et al. 2013]. For instance, Mbed-TLS generates AES S-box during execution instead of storing them as hard-coding values in the data section.[1] Signsrch then fails to detect AES in Mbed-TLS. On the other hand, operation distribution is a typical pattern of cryptographic implementations. For example, ReFormat [Wang et al. 2009] observes the ratio of arithmetic and bitwise operations to infer if a function is relevant to cryptography. However, it can hardly decide the specific algorithm that the function implements.

2.1.2 Graph-Based Techniques. Since cryptographic primitives are sets of arithmetic and logic operations, the structural relationships between the data and operations are considered to be consistent in binaries. WheresCrypto [Meijer et al. 2021] and the technique proposed by Lestringant et al. [Lestringant et al. 2015] then base the identification on the isomorphism analysis

[1] Dynamic generation of AES S-box is a well-established research field to protect the implementations from modern cryptanalysis attacks, such as side-channel attacks [Ashokkumar et al. 2018; Singh et al. 2017].



Function name	Predicate	node	Patterns found
_nettle_camellia_crypt	(R3&CONST:f) ...	2	
_nettle_camellia_crypt	(R3&CONST:f) ...	1	
_nettle_camellia_crypt	(R3&CONST:f) ...	381	AES (encrypt) (Variant A), AES (decrypt) (Variant A)
_nettle_camellia_crypt	(R3&CONST:f) ...	660	Sequential Block Permutation
_nettle_camellia_crypt	(R3&CONST:f) ...	1674	Sequential Block Permutation
_nettle_camellia_crypt	(R3&CONST:f) ...	2843	Sequential Block Permutation
_nettle_camellia_crypt	(R3&CONST:f) ...	1748	Sequential Block Permutation
_nettle_camellia_crypt	(R3&CONST:f) ...	7515	Sequential Block Permutation

Fig. 1. WheresCrypto's Results on *nettle_camellia_crypt* from `libnettle` v3.9.1

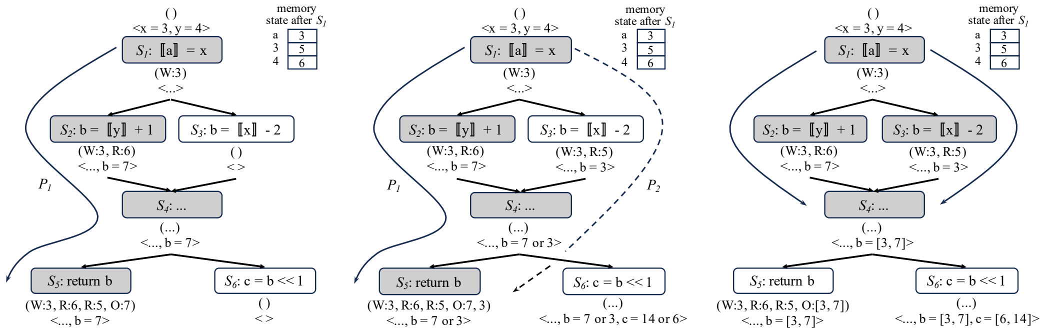
of the data flow graphs (DFGs). Unfortunately, data flow relationships are still vulnerable to code transformation. For example, compiler optimization might change code representations and structures tremendously [Chandramohan et al. 2016; Wang et al. 2023; Wang and Wu 2017; Xue et al. 2018], which alters those of DFG accordingly. Figure 1 presents the results of WheresCrypto to analyze the core function of `Camellia` (*nettle_camellia_crypt*), a Feistel cipher [Aoki et al. 2000], from `Nettle` [Möller 2013] which is compiled with GCC-O3. It fails to find the Feistel structure in the function, only considering it as the sequential block permutation. Furthermore, it misidentifies the function as `AES`.

2.1.3 Execution-Based Techniques. Cryptographic algorithms usually employ lots of loops for implementation, because they largely depend on repeated data transformation [Lutz 2008]. Then, the execution-based techniques are proposed to capture fine-grained semantics from execution traces, leveraging runtime data in iterations to identify cryptographic implementations, e.g., CryptoHunt [Xu et al. 2017b] and Aligot [Calvet et al. 2012]. Crafted inputs are the preliminary of these techniques such that they are able to cover the target code. The limited code coverage hinders their practical application. Besides, dynamic instrumentation is heavy-weight, and trace analysis is time-consuming due to the substantial amount of data collected during loop execution. For instance, CryptoHunt introduces a 5-6X slowdown for online trace logging and spends 30.8 minutes on offline analysis to find `AES` in `OpenSSL` [Xu et al. 2017b].

2.2 Challenge: Practicality of the Analysis

The existing techniques identify binary cryptographic functions generally relying on similarity analysis. They prepare reference implementations beforehand, extract features from the target code under analysis, and compare the similarity between the features and those of the references. Unfortunately, it is challenging to achieve the balance between *accuracy*, *efficiency*, and *code coverage* of the analysis, rendering the existing techniques impractical.

2.2.1 Accuracy. Optimization is the core step of compilation to generate binary code, and obfuscation is a common practice for code protection. They are both semantics-preserving but alter the representation and structures of the code significantly. In addition, even the source code of the same algorithm could be implemented quite differently, which further leads to differences in the binaries. As such, it is difficult to *accurately* perform the comparison between the target code and references on the binary level. The static techniques, e.g., those based on heuristics and graphs,



(a) Single-path Emulation (b) Multiple-path Emulation (c) Path-insensitive Emulation

Fig. 2. Examples of code emulation and feature extraction. The branching statements in the control flow graphs are omitted. At each program point, (·) and <·> contains the code features and emulated program state respectively. W, R, and O represents written value, read value, and outputted value, separately. ⟦·⟧ in a graph node means memory accessing.

extract features from assembly code or code graphs, which can hardly handle optimization and obfuscation.

2.2.2 *Efficiency and Coverage.* Intuitively, for better accuracy, it is necessary to gather more expressive code features that are closely related to semantics. However, that requires covering all the code and learning code behaviors, which is time-consuming. Thus, it is tough to obtain the code semantics *efficiently* with the *complete coverage.* Although the dynamic techniques, e.g., those based on actual execution, can obtain information about code behaviors, they still suffer from the overhead of online instrumentation and limited code coverage.

2.3 Basic Idea of BinCrypto

BinCrypto identifies binary cryptographic functions via similarity analysis with path-insensitive emulation on the interval domain. The insight is that two pieces of semantics-equivalent code would generate similar output if provided with the same input. It is inspired by the Schwartz–Zippel lemma to solve the problem of polynomial identity testing (PIT), which aims to determine whether two polynomials are equal [Egele et al. 2014; Saxena 2009, 2014]. The lemma states that, given a randomly chosen input, the probability is low that two polynomials yield equal results if they are inequivalent [Schwartz 1980; Zippel 1979]. Hence, given the same input, semantics-equivalent code should produce similar output, even though the input is composed of random values.

2.3.1 *Accuracy: Code Emulation for IO Values.* Considering the target function and reference cryptographic function, BinCrypto adopts emulation to interpret them with the same input. During the process, IO values are collected as code features for accurate similarity analysis. Features of IO values have been shown to be effective for cryptographic function identification [Calvet et al. 2012; Li et al. 2012]. As long as the target function implements the same algorithm as the reference one, their emulated IO values should be similar, even though they are varied in code representations and structures.

Technically, the emulation adopted by BinCrypto aims to reason about program behaviors via static code interpretation. It can be started at any program point with arbitrary inputs, obtaining possible variable values triggered by the inputs. Figure 2a depicts how the emulation works.

Assuming the input is $< x = 3, y = 4 >$, the emulated execution triggers the path P_1, i.e., $S_1 \rightharpoonup S_2 \rightharpoonup S_4 \rightharpoonup S_5$, and IO values are collected accordingly. Specifically, S_1 writes x's value 3 into memory at address a. BinCrypto then records $W : \mathit{3}$ as a feature of Writting. Similarly, it finds S_2 reads value *6* from memory, then S_5 returns b's value 7 as the output, achieving the feature sequence of $(W : \mathit{3}, R : \mathit{6}, O : \mathit{7})$ at last, where R and O mean features of Reading and Outputting separately.

2.3.2 Efficiency and Coverage: Interval-Based Path-Insensitive Analysis. BinCrypto performs the emulation path-insensitively on the interval domain to achieve the complete block coverage efficiently. Originally, for a single input, the emulation covers only one path, as shown in Figure 2a. Since it can be started at any program point, for complete coverage, a naive solution is to conduct the single-path emulation multiple times until all the code is covered, which however would lead to state explosion.

Figure 2b shows the example of multiple-path emulation. After processing the path P_1 in the first round, S_3 becomes the target to be covered next. With the inherited program state from S_1, the emulation is started from S_3 and triggers the path P_2,[1] i.e., $S_1 \rightharpoonup S_3 \rightharpoonup S_4 \rightharpoonup S_5$,[2] finding that S_3 reads value *5* from address *3* and b gets another value *3* as the outputting feature. Lastly, S_6 is emulated for the complete code coverage with the program state after processing S_4 as the input. Since b has two possible values, i.e., *7* or *3*, S_6 has to be emulated twice to complete the analysis. Consequently, as the emulation goes deeper, the issue of state explosion would occur to hamper the efficiency of the analysis.

To this end, BinCrypto traverses the code in a path-insensitive manner. At each branching point, it emulates each succeeding path respectively and merges the emulated program states of all the paths at the following post-dominator. The variable values are also joined and represented as intervals, which over-approximate the possible values of those variables under the given random input. As such, BinCrypto is able to process each basic block only once, avoiding the explosion of emulated program states and saving analysis time. Functions implementing the same cryptographic algorithms should exhibit more correlated features than those of inequivalent semantics.

Figure 2c presents the interval-based path-insensitive emulation of BinCrypto. Starting from S_1, it traverses S_2 and S_3 separately and merges the two program states at S_4, the immediate post-dominator of S_1. The emulated values are then represented with intervals, e.g., $b = [3, 7]$ at S_4. Similarly, S_5 and S_6 are processed respectively, with the program state after the emulation of S_4, ensuring that every block is covered and processed only once. The emulated variable values are represented as intervals accordingly as well.

2.4 System Overview of BinCrypto

Given the target binary file, i.e., an executable or a library, BinCrypto extracts IO behaviors as features of each target function via path-insensitive code emulation. The features are then compared to those of the reference cryptographic functions to calculate similarity scores. The higher the score is, the more likely it is that the target function implements the same algorithm as the reference one. The target function with the highest score is considered most likely to implement the same algorithm as the reference one.

Figure 3 presents the workflow of BinCrypto. It first disassembles the target binary to achieve the assembly code and control flow graph (CFG) of each function. Then, it captures the static information of the disassembled code via *Preprocessing* (§3.2), including loop identification and parameter recognition. Afterwards, provided with the seed input, which is composed of random

[1] The emulation of BinCrypto can be started from any program point with arbitrary inputs. It disregards the path constraints and covers S_3 by force, taking the program state after S_1 as the input.

[2] Due to the inheritance of the state from S_1, the actual processed nodes are S_3, S_4, and S_5 in this round.

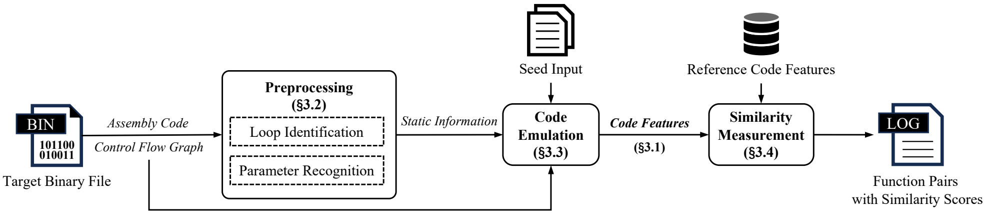


Fig. 3. System Overview of BinCrypto

values generated beforehand, BinCrypto performs *Code Emulation* (§3.3) on the CFG of each function along with the static information to extract its *Code Features* (§3.1). At last, The features are compared to those of the reference cryptographic functions, which are also derived with the same seed input, to compute similarity scores, i.e., *Similarity Measurement* (§3.4).

3 Design

In this section, we first introduce the code features adopted by BinCrypto. Then, we explain how it extracts and processes the features in detail.

3.1 Code Features

Input and output values reflect code behaviors, which are effective features for cryptographic function identification [Calvet et al. 2012; Li et al. 2012]. BinCrypto then collects the following emulated input and output values as code features.

1) ***Input Values.*** For a binary function, in addition to the arguments, its input values derive from the following sources [Zhang and Qian 2018], including:

- *Global Variables.* Their values are stored in data sections, e.g., *.data*, *.rodata*, etc.
- *Heap Variables.* The values are stored in memory dynamically allocated by library functions, including *malloc*, *calloc*, *realloc*, etc.
- *Return Values of Library Functions.* According to calling conventions, a return value is stored in a specific register of an instruction set architecture, e.g., *RAX* for an integer on *x64*. Additionally, Symbol names of library functions persist in stripped binaries because they are necessary for resolving external function invocations. Then, *symbol names* of library functions invoked during the emulation are also considered as features.

2) ***Output Values.*** Output values contain those written as *Global* and *Heap Variables*. *Return Value* of a user-defined function under analysis is considered as output as well.

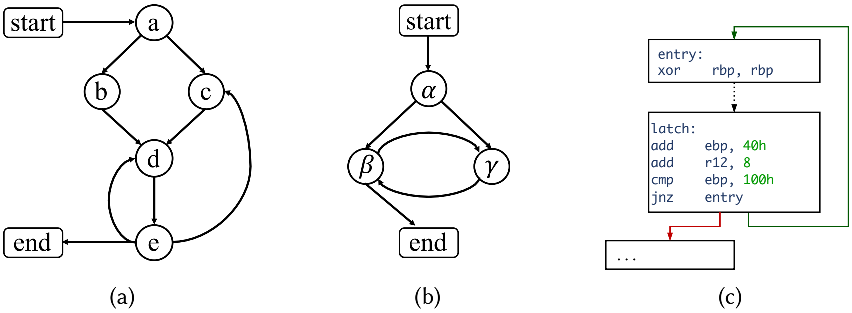
3.2 Preprocessing

BinCrypto captures the static information of each binary function, which is essential for later emulation.

3.2.1 Loop Identification. Since the seed input values are generally illegal for actual execution, the emulation would be trapped in loops easily. Thus, it is necessary to identify such structures beforehand to facilitate loop emulation (§3.3). In this step, BinCrypto figures out the back-edges and entries of a loop.[1]

Irreducible Loop Analysis. A loop is irreducible if it has multiple entries [Havlak 1997]. It appears because of the adoption of GOTO in the source code and compiler optimization. The choice of entry

[1] Back-edge is an edge that points to a block that has already been met during a depth-first traversal of the control flow graph. The destination block is a loop entry [Havlak 1997].



```
entry:
xor    rbp, rbp

latch:
add    ebp, 40h
add    r12, 8
cmp    ebp, 100h
jnz    entry

...
```

Fig. 4. Samples of Irreducible and Regular Loops

could be arbitrary based on the depth-first traversal order on the control flow graph (CFG) [Sreedhar et al. 1996; Unger and Mueller 2002]. For example, in Figure 4a, $L = \{c, d, e\}$ is an irreducible loop. Node c is considered as the entry if the depth-first traversal path is $a \rightharpoonup \underline{c \rightharpoonup d \rightharpoonup e}$ then $a \rightharpoonup b$. By contrast, Node d is the entry if the path is $a \rightharpoonup b \rightharpoonup \underline{d \rightharpoonup e \rightharpoonup c}$.

BinCrypto chooses the node that has a shorter (longer) distance to the `start` (end) node in the CFG as the entry of an irreducible loop. The distance between two nodes is the number of edges in the shortest path connecting them [West et al. 2001]. In Figure 4a, Node c is chosen as the loop entry, because it is closer to the `start` node than d. If the entries share the same distance to the `start` node, BinCrypto checks their distances to the end node next. As depicted in Figure 4b, Node γ is then chosen as the entry instead of β, because it is farther away from end. In the worst case, if there is an edge from γ to end in Figure 4b, BinCrypto then treats β and γ both as entries of the loop.

Regular Loop Detection. A loop is regular if it is iterated with a fixed number of times [Wolfe et al. 1995]. With the definite number of iterations, a regular loop provides a good chance for code optimization, such as loop unrolling. BinCrypto identifies regular loops such that it is able to process them in a different way from irregular ones in the next stage. Since heuristics are hardly avoidable for reverse engineering [Lin and Gao 2021; Liu et al. 2023; Wang et al. 2017, 2015], BinCrypto leverages the pattern of operations to achieve the goal. A loop is inferred to be regular if it is found that the source block of the loop back-edge contains a variable: i) which is increased or decreased by a constant value; and ii) then is compared to another constant, deciding the jump target next. For example, Figure 4c presents a regular loop with the loop body omitted. In the source block of its back-edge (`latch`), the variable *ebp* is firstly increased by *0x40*. Then, it is compared with *0x100*, to decide if jumping back to the entry. Thus, that is a regular loop with *4* iterations.

3.2.2 Parameter Recognition. Argument values constitute the critical component of input. Since modern 64-bit instruction set architectures are commonly rich in registers, they pass function arguments via specific registers as well as stack if those registers are used up. Thus, to guarantee the same input for emulation, it is necessary to recognize parameters in order to facilitate argument assignment afterwards (§3.3.3 and Line 2-3 in Algorithm 3).

BinCrypto traverses the function CFG in a depth-first manner, recognizing the registers and stack variables as function parameters if they are used before defined. The process is presented in Algorithm 1. For a candidate parameter a_c, i.e., either stored in a parameter register or in the parameter area in the stack frame, BinCrypto considers it to be used if it participates in computing the output. Specifically, if a variable itself or another one, which depends on it, is

Algorithm 1: Used Parameter Recognition

```
   Input: Control Flow Graph of the Function $G$
   Input: Set of Candidate Parameters of the Function $A$
   Output: Used Argument Set $S$
1  Algorithm RecognizeUsedParameter (G, A)
2      foreach candidate parameter $a_c \in A$ do
3          foreach node n in G do
4              foreach instruction i in n do
5                  if i performs local data movement then continue
6                  if any variable accessed by i depends on $a_c$ then
7                      add $a_c$ to $S$
8                  else if i calls a subroutine r then
9                      $s \leftarrow$ RecognizeUsedParameter ($G_r$, $A_r$)
10                     if $\exists a_r \in s$ depends on $a_c$ then add $a_c$ to $S$
11     return $S$
```

accessed by an instruction (e.g., memory accessing, arithmetic or logic computation, etc.), the variable is considered to be **used** (Line 6-7). Local data movement operations is excluded (Line 5) because the operation does **not** compute the output directly [Cho et al. 2019]. On the other hand, a variable is **used** if its value is passed as an argument of the subroutine, which is invoked by the current function (Line 8-10).

Besides, since the factors involved in cryptographic algorithms are determined by the specific algorithm (e.g., plaintext, ciphertext, keys, nonces, etc.), the probability is low to implement them with variadic functions in practice.[1] Thus, BinCrypto finds variadic functions during parameter recognition to exclude them for similarity analysis, which improves the efficiency. The recognition is performed based on the patterns of argument management for the specific architecture. According to `System V AMD64 ABI` [Lu et al. 2018], to access arguments in a variadic list, all the arguments passed by registers should be pushed onto the stack, including the floating-point (FP) ones. Another two variables are specified on the stack to indicate the stack location of the start of the register saved area (*reg_save_area*) and the first argument passed on the stack (*overflow_reg_area*). In practice, due to code optimization, a compiler might discard the movement of FP arguments if it ensures that the function cannot have such type of parameters. Thus, there is only code to process integer register arguments and initialize *reg_save_area* along with *overflow_reg_area*. Based on such patterns, BinCrypto is able to detect variadic functions.

3.3 Code Emulation

In this section, we first introduce the language adopted to facilitate the explanation of the method. Then, we demonstrate how BinCrypto emulates the instructions of the language and manipulates the control flow accordingly.

3.3.1 Language. For clarity, we adopt a C-like language to describe the function under emulation, as presented in Figure 5. A binary function is composed of a sequence of instructions, which generally could be classified into three categories in terms of the functionality: i) *data movement* instructions copy data items between registers and memory locations; ii) *arithmetic and logic*

[1] In our datasets (§5.1.1), no standard cryptographic algorithms are found to be implemented with variadic functions.

$$\begin{array}{llcll}
\textit{Function} & \mathcal{F} & \coloneqq & \mathcal{I} & \\
\textit{Instruction} & \mathcal{I} & \coloneqq & r_1 \leftarrow r_2 & \textbf{assign} \\
 & & \mid & r_1 \leftarrow r_2 \oplus r_3 & \textbf{arithmetic} \\
 & & \mid & r_1 \leftarrow r_2 \otimes r_3 & \textbf{logic} \\
 & & \mid & r_1 \leftarrow r_2 \ominus r_3 & \textbf{comparison} \\
 & & \mid & r_1 \leftarrow \llbracket r_2 \rrbracket & \textbf{read} \\
 & & \mid & \llbracket r_1 \rrbracket \leftarrow r_2 & \textbf{write} \\
 & & \mid & \mathcal{I}_1 ; \mathcal{I}_2 & \textbf{sequence}
\end{array}$$

$$\oplus \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{...}\} \quad \otimes \in \{\texttt{\&},\texttt{|},\texttt{\^{}},\texttt{...}\} \quad \ominus \in \{\texttt{<},\texttt{=},\texttt{>},\texttt{...}\}$$

Fig. 5. Syntax of the Language

$$\frac{\mathbb{S}(r_2) = [l_2, u_2]}{\mathbb{S} \vdash r_1 \leftarrow r_2 : \mathbb{S} \cup \{(r_1, \mathbb{S}(r_2))\}}\ \textbf{Assign} \qquad \frac{\mathbb{S} \vdash I_1 : \mathbb{S}_1 \qquad \mathbb{S}_1 \vdash I_2 : \mathbb{S}_2}{\mathbb{S} \vdash I_1; I_2 : \mathbb{S}_2}\ \textbf{Sequence} \qquad \frac{\text{address } r_1 \qquad \mathbb{S}(r_2) = [l_2, u_2]}{\mathbb{S} \vdash \llbracket r_1 \rrbracket \leftarrow r_2 : \mathbb{S} \cup \{(\llbracket r_1 \rrbracket_{\mathbb{S}}, \mathbb{S}(r_2))\}}\ \textbf{Write}$$

$$\frac{\mathbb{S}(r_2) = [l_2, u_2]}{\mathbb{S} \vdash r_1 \leftarrow \neg r_2 : \mathbb{S} \cup \{(r_1, [\neg u_2, \neg l_2])\}}\ \textbf{BitwiseNot} \qquad \frac{\text{address } r_2 \qquad \text{random value } r_v}{\mathbb{S} \vdash r_1 \leftarrow \llbracket r_2 \rrbracket : \begin{cases} \mathbb{S} \cup \{(r_1, r_v), (\llbracket r_2 \rrbracket_{\mathbb{S}}, r_v)\} & \textit{if } r_2 \textit{ not in } \mathbb{S} \\ \mathbb{S} \cup \{(r_1, \llbracket r_2 \rrbracket_{\mathbb{S}})\} & \textit{otherwise} \end{cases}}\ \textbf{Read}$$

$$\frac{\text{operand } r_2 \qquad \text{operand } r_3}{\mathbb{S} \vdash r_1 \leftarrow r_2 \mid r_3 : \mathbb{S} \cup \{(r_1, [\lfloor r_2 | r_3 \rfloor, \lceil r_2 | r_3 \rceil])\}}\ \textbf{BitwiseOr} \qquad \frac{\mathbb{S}(r_2) = [l_2, u_2] \quad \mathbb{S}(r_3) = [l_3, u_3]}{\mathbb{S} \vdash r_1 \leftarrow r_2 \oplus r_3 : \mathbb{S} \cup \{(r_1, [l_2 \oplus l_3, u_2 \oplus u_3])\}}\ \textbf{Add/Mul}$$

$$\frac{\mathbb{S}(r_2) = [l_2, u_2] \qquad \mathbb{S}(r_3) = [l_3, u_3]}{\mathbb{S} \vdash r_1 \leftarrow r_2 \ominus r_3 : \begin{cases} \mathbb{S} \cup \{(r_1, \texttt{True})\} & \textit{if } l_2 + u_2 \ominus l_3 + u_3 \\ \mathbb{S} \cup \{(r_1, \texttt{False})\} & \textit{otherwise} \end{cases}}\ \textbf{Less/LessOrEqual}$$

$$\textit{Interval } t = [l, u] = \{v \mid l \leq v \leq u,\ \textit{where } v, l, u \in Z\}$$

Fig. 6. Rules for instruction emulation with intervals. $\mathbb{S}$ represents the emulated program state mapping a variable to its interval. $\llbracket \cdot \rrbracket_{\mathbb{S}}$ means memory accessing from $\mathbb{S}$. $\lfloor \cdot | \cdot \rfloor$ and $\lceil \cdot | \cdot \rceil$ computes lower and upper bound values for **BitwiseOr**.

instructions implement the computation of binary code; iii) *control flow* instructions manipulate the control flow during code execution.

The language contains assignments, memory reading, and writing for data movement. Arithmetic and logic operations are covered as well. The control flow instructions primarily consist of conditional comparisons, function calls and returns, and explicit/implicit jumps. Comparison operations are involved in the language, which are the basis for implementing branches and loops in addition to their normal use. Jumps are simple just for control flow transferring, which are then omitted for discussion simplicity. Function call and return instructions essentially are jumps with data movement between the caller and callee. Since BinCrypto is inter-procedural, the two instructions are also omitted. Note that the language is adopted to demonstrate how BinCrypto performs code emulation on intervals to overcome the challenges described in Section 2.3. It abstracts away other common operations, such as unary instructions, which can be handled straightforwardly in practice.

3.3.2 *Instruction Emulation on Intervals.*

Figure 6 presents the inference rules for emulating instructions specified in the language. Each rule updates the emulated program state $\mathbb{S} : r \mapsto [l, u]$, which maps a variable to its interval. The variables above the horizontal line constitute the preliminaries of the rule. Those under the line depict the program states before and after emulating the given instruction $\mathcal{I}$, denoted as $\mathbb{S} \vdash \mathcal{I} : \mathbb{S}'$.

Data Movement. The assignment and write rules are straightforward, which update the variable values accordingly. The emulation of memory reading might cause errors, because the seed input values are generally illegal values for actual execution, e.g., reading the memory pointed by the

Algorithm 2: Bounds Computation for BitwiseOr

```
   Input: Lower and Upper Bounds of the 1st Interval $l_1$, $u_1$
   Input: Lower and Upper Bounds of the 2nd Interval $l_2$, $u_2$
   Output: Lower and Upper Bounds of the BitwiseOr Result
1  Algorithm ComputeLowerBound ($l_1, u_1, l_2, u_2$)
2      $mask \leftarrow 1 \ll \texttt{max}(l_1\text{'s bit number}, l_2\text{'s bit number})$
3      $l \leftarrow l_1 \hat{} \, l_2$
4      while $mask \neq 0$ do
5          if $l \,\&\, mask \neq 0$ then
6              if $l_1 \,\&\, mask = 0$ then
7                  $t \leftarrow (l_1 \mid mask) \,\&\, (\neg mask + 1)$
8                  if $t \leq u_1$ then $l_1 \leftarrow t$, break
9              else if $l_2 \,\&\, mask = 0$ then
10                 $t \leftarrow (l_2 \mid mask) \,\&\, (\neg mask + 1)$
11                 if $t \leq u_2$ then $l_2 \leftarrow t$, break
12         $mask \leftarrow mask \gg 1$
13     return $l_1 \mid l_2$
14 Algorithm ComputeUpperBound ($l_1, u_1, l_2, u_2$)
15     $mask \leftarrow 1 \ll \texttt{max}(u_1\text{'s bit number}, u_2\text{'s bit number})$
16     while $mask \neq 0$ do
17         if $u_1 \,\&\, u_2 \,\&\, mask$ then
18             $t \leftarrow (u_1 - mask) \mid (mask - 1)$
19             if $t \geq l_1$ then $u_1 \leftarrow t$, break
20             $t \leftarrow (u_2 - mask) \mid (mask - 1)$
21             if $t \geq l_2$ then $u_2 \leftarrow t$, break
22         $mask \leftarrow mask \gg 1$
23     return $u_1 \mid u_2$
```

argument of value *0xdeadbeef*. If the reading address is illegal, i.e., not involved in the program state $\mathbb{S}$, the memory location is initialized with a random value for returning (r_v).

Arithmetic and Logic Computation. The addition and multiplication rules describe how to process basic arithmetic operations, based on which subtraction and division could be inferred. The rule of BitwiseNot is easy to understand. For BitwiseOr, given two intervals $[a, b]$, $[x, y]$, the naive result of the operation is $[\texttt{max}(a, x), b + y]$, which introduces imprecision.

Algorithm 2 describes how BinCrypto computes the lower and upper bound of the bitwise or operation between two intervals. For the two lower bound values l_1 and l_2, *ComputeLowerBound*[1] performs scanning starting from the most significant bit (Line 2), trying to increase them to achieve the maximum value of the resultant lower bound. At the same position, if the bit values are both *0* or *1*, the resultant lower bound would have *0* or *1* at that position as well, and the scanning continues (Line 12). Otherwise, when the two bit-values are different, assuming l_1 has *0* and l_2 has *1*, *ComputeLowerBound* flips that *0* of l_1 into *1* and unsets all its following bits (Line 7 and 10). If that value is valid, i.e., still belonging to the interval, l_1 is updated accordingly (denoted as l_1'), and the scanning ends (Line 8 and 11), with $l_1' \mid l_2$ giving the result (Line 13). *ComputeUpperBound* works in the same way. The difference is that it aims to find the minimum value of the resultant

[1] Its correctness is shown in the extended version of this paper [Hu et al. 2025].

$$\begin{array}{ccccccccccccc} & & b_3 & b_2 & b_1 & b_0 & & b'_3 & b'_2 & b'_1 & b'_0 & & \\ l_1 & [& 0 & 0 & 1 & 0 & , & 0 & 0 & 1 & 1 &] & u_1 \\ l_2 & [& 0 & 1 & 0 & 1 & , & 1 & 0 & 1 & 0 &] & u_2 \\ l'_2 & & 0 & 1 & 1 & 0 & & 1 & 0 & 0 & 1 & & u'_2 \end{array}$$

Fig. 7. Bounds Computation for BitwiseOr

upper bound. Thus, when the two bits of u_1 and u_2 are both *1* at the same position, one *1* is flipped into *0* and the subsequent bits are all set then (Line 18 and 20).

Figure 7 depicts an example of computing the bounds values of BitwiseOr for intervals $[2, 3]$ and $[5, 10]$, where the values have been represented in binaries. For the lower bound, l_1's b_2 is different from l_2's. However, the change cannot be applied to l_1, because the candidate value of l'_1 is *0b0100*, which is greater than u_1. By contrast, b_1 of l_2 could be flipped, generating l'_2 as *0b0110*. Therefore, the resultant lower bound is *6* (=*0b0010* | *0b0110*). For the upper bound, b'_1 of u_2 is found to be valid for flipping, generating *0b1001* as u'_2. Then the resultant upper bound is *11* (=*0b0011* | *0b1001*). $[6, 11]$ is tighter than the naive result of $[5, 13]$. Additionally, the bounds values of BitwiseAnd could be inferred from BitwiseNot and BitwiseOr according to De Morgan's laws [Copi et al. 2016]. Then, those of BitwiseXor are handled based on BitwiseOr and BitwiseAnd.

Sequence and Comparison. The Sequence rule states that the consecutive instructions are processed in order. The Less and LessOrEqual rule shows that BinCrypto compares two intervals by checking their average values.[1] The > and ≥ relations are transformed into ≤ and < accordingly. The = and ≠ relations are handled by checking whether the bounds values of two intervals are equal or not. Comparison is the basis of branching. To achieve complete block coverage, BinCrypto covers both the `True` and `False` branches through path-insensitive emulation (as shown in Figure 2c). The features of IO values are recorded in a sequence that adheres to the emulation order of instructions. Guided by the Less and LessOrEqual rule, BinCrypto obtains the outcome of comparing two intervals to determine the order in which to emulate the two branches next. The details are described in Section 3.3.3 (Branches).

3.3.3 Control Flow Manipulation. Algorithm 3 shows the processes of BinCrypto to manipulate the emulated control flow. Given a control flow graph (CFG), it starts from the given start block and keeps traversing until reaching the given end block, covering all blocks on the paths between them. If the start block is the entry of a function, which is not invoked by the process, BinCrypto initializes its arguments with the seed input (Line 2-3).

Function Calls and Jumps. For a library function call, BinCrypto updates the emulated program state for the memory allocation, e.g., *malloc* (Line 8-10). Other library functions, which do not require system support, e.g., *memcpy*, are emulated based on the current program state according to their symbol names (Line 11).

Function inlining is a common optimization of modern compilers [Chandramohan et al. 2016]. When a user-defined function is invoked, BinCrypto then tries to infer whether it could be inlined. If that is the case, BinCrypto steps into the function and performs the emulation recursively (Line 13-15), bridging the gaps between optimized and unoptimized code. Otherwise, for the sake of efficiency, it only covers one path of the callee to derive the return value (Line 16).

In addition, if the instruction is a jump with an illegal target, e.g., an indirect jump, BinCrypto just skips it and continues the emulation with the next instruction (Line 17-18). If the instruction

[1] $l_2 + u_2 \ominus l_3 + u_3 \equiv \frac{1}{2}(l_2 + u_2) \ominus \frac{1}{2}(l_3 + u_3)$.

```
Algorithm 3: Control Flow Manipulation
   Input: Start / End Node on the CFG for Emulation $N_s$ / $N_e$
   Input: Emulated Program State $S$
   Output: Function Code Feature Set $F$
 1 Algorithm Emulate ($N_s$, $N_e$, $S$)
 2     if $N_s$ is function $f$'s entry and $f$ is not a callee then
 3         initialize $f$'s arguments with the seed input
 4     $b \leftarrow N_s$
 5     while $b \neq N_e$ do
 6         foreach instruction $i$ in $b$ do
 7             if $i$ will invoke a library function $f$ then
 8                 if $f$ allocates memory $m$ dynamically then
                       // e.g., malloc, calloc, and realloc
 9                     $m_{st} \leftarrow$ a random value
10                     allocate $m$ in $S$ starting at $m_{st}$
11                 else $f$ is handled according to its symbol name
12             else if $i$ will invoke a callee $c$ then
                   // inline $c$ for more code features
13                 if $c$ could be inlined then
14                     $c_s, c_e \leftarrow$ GetStartEndBlock ($c$)
15                     $F \leftarrow F \cup$ Emulate ($c_s$, $c_e$, $S$)
16                 else $F \leftarrow F \cup$ EmulateSinglePath ($c$, $S$)
17             else if $i$ is a jump with illegal target then
18                 continue
19             else $F \leftarrow F \cup$ EmulateInstruction ($i$, $S$)
20         if $b$ is the entry of a loop $l$ then
21             $k \leftarrow$ the number of times to unroll a loop
22             $F \leftarrow F \cup$ EmulateLoop ($l$, $k$, $S$)
23         else if $b$ has multiple successors then
24             $C_o \leftarrow S$
25             $ipdom \leftarrow$ GetImmediatePostDominator ($b$)
26             foreach successor $s$ of $b$ with specific order do
27                 $C_t \leftarrow C_o$
28                 $F \leftarrow F \cup$ Emulate ($s$, $ipdom$, $C_t$)
                   // merge program states for branches or switches
29                 $S \leftarrow S \sqcup C_t$
30         $b \leftarrow$ GetNextUncoveredBlock ( )
31     return $F$
```

is control-flow irrelevant, it then handles the instruction according to the rules and updates the program state accordingly (Line 19).

Loops. If the current block is the entry of a loop, the emulation is considered to be in a loop structure (Line 20). BinCrypto unrolls the loop for a pre-defined number of times to cover all the code blocks [Biere et al. 1999]. Note that, for a regular loop with fixed number of iterations (§3.2), BinCrypto unrolls it for that number to better uncover its behaviors (Line 21-22).

Table 1. Example of LCS table for interval-based sequence similarity measurement. Items of the same color are aligned. The number in each cell represents the LCS length of the interval sequences.

X Y	ϵ	$[0, 3]$	$[5, 6]$	$[1, 8]$
ϵ	0	0	0	0
$[2, 5]$	0	1/3	1/3	1/2
$[0, 1]$	0	1/2	1/2	1/2
$[3, 7]$	0	1/2	9/10	9/8

Branches. In addition to the source block of a loop back-edge, a basic block can have multiple successors because it ends with a conditional jump or it dispatches the control flow to implement a switch structure. BinCrypto first finds the immediate post-dominator (*ipdom*) of the block with multiple successors (Line 25). Then, it performs the emulation from each successor block to the *ipdom*, and merges the program states to process the following block (Line 28-29).

The successors are enumerated in a specific order such that the feature sequences collected from equivalent code follow the same order. For a conditional branch, BinCrypto unifies the comparison into its equivalent form. Specifically, it transforms *Greater than* (>) into *Less than or Equal to* (≤), *Greater than or Equal to* (≥) into *Less than* (<), and *Not Equal to* (≠) into *Equal to* (=). After that, it emulates the `True` branch then the `False` branch in order. For a switch structure, BinCrypto would process each case with the same order as that in the jump table. As a result, BinCrypto processes each block only once, achieving the complete block coverage efficiently.

3.4 Similarity Measurement

During the emulation, BinCrypto records the emulated IO values to form a feature sequence for each target function,[1] which is then compared to those of the reference functions. Given two sequences of code features X, Y, their similarity score is calculated with the Jaccard Index:

$$Score = \mathtt{J}(X,\ Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|},$$

where $X \cap Y$ is the longest common subsequence (LCS) of the two sequences. Since the feature elements are represented with intervals, which also could be treated as sets of integers, BinCrypto measures the similarity of two intervals with the Jaccard Index as well. Then, based on the conventional algorithm [Wagner and Fischer 1974], it computes the LCS table T for sequence $X[1 \ldots i]$ and $Y[1 \ldots j]$ as follows:

$$T_i^j = \begin{cases} 0 & if\, i = 0 \ or\ j = 0 \\ max(T_{i-1}^{j-1} + \mathtt{J}(X_i,\ Y_j),\ T_{i-1}^{j},\ T_i^{j-1}) & if\, X_i \cap Y_j \neq \emptyset \\ max(T_{i-1}^{j},\ T_i^{j-1}) & if\, X_i \cap Y_j = \emptyset \end{cases},$$

where X and Y are two feature sequences, and X_i, Y_j means their i-th and j-th element separately, which are intervals.

Example. Table 1 shows the LCS table to measure the similarity of interval-based sequences. X and Y has 3 intervals in sequence separately. Their LCS is achieved when X's $[0, 3]$ and $[1, 8]$ are aligned with Y's $[0, 1]$ and $[3, 7]$ respectively, where the similarity scores are *0.5* ($= \frac{1}{2} = \frac{|[0,3] \cap [0,1]|}{|[0,3] \cup [0,1]|}$) and

[1] The sequence follows the emulation order of the corresponding instructions which manipulate IO values.

Table 2. Cryptographic Libraries Adopted in the Evaluation

Library	Version	LoC	Number of Cryptography Related Functions
GnuTLS	3.8.0	633,991	459
cryptolib	3.4.7	68,782	158
Mbed-TLS	3.4.0	65,529	365
wolfSSL	5.6.3	97,949	365
Libgcrypt	1.8.10	30,483	339
Nettle	3.9.1	15,504	988
libfreeblpriv3[1]	3.92	14,500	414
LibreSSL	3.7.3	22,280	1,041
libcrypto[2]	1.1.1f	16,670	960

[1] component of NSS to handle cryptographic operations
[2] component of OpenSSL to provide cryptographic functions

0.625 ($= \frac{5}{8} = \frac{|[1,8] \cap [3,7]|}{|[1,8] \cup [3,7]|}$). Then, the LCS length of X and Y, i.e., $|X \cap Y|$, is *1.125* ($= \frac{9}{8} = 0.5 + 0.625$), and $|X \cup Y|$ is 4.875 ($= 3 + 3 - 1.125$). As a result, the similarity score of X and Y is *0.231* ($=\frac{1.125}{4.875}$).

4 Implementation

The prototype of BinCrypto supports binary cryptographic function identification for ELF files with x64 instruction set architecture on the Linux platform. It adopts `IDA Pro` [Hex-rays 2024] to perform disassembling. Based on `IDAPython`, about *3,600* lines of Python code is developed to automate preprocessing (§3.2). The emulation is built based on `QEMU` [Bellard 2005] with around *15,000* lines of C/C++ code to enable path-insensitive traversal (§3.3). `IDA Pro` and `QEMU` are adopted due to their widespread utilization and sophisticated support for research. While the two tools are not perfect [Flores-Montoya and Schulte 2020; Miller et al. 2019; Quynh and Vu 2015; Zhang et al. 2021], they are adequate for the prototype to demonstrate the effectiveness of BinCrypto. The inlining inference adopts the idea of selective inlining [Chandramohan et al. 2016; Xue et al. 2018]. Regular loops with fixed number of iterations are unrolled for that number (§3.2), while others are unrolled once.

5 Evaluation

In this section, we conduct empirical experiments to evaluate the effectiveness and capacity of BinCrypto with the following research questions (RQs):

- **RQ1**: What is its performance in identifying cryptographic functions in real-world libraries generated with diverse compilation settings (§5.2)?
- **RQ2**: How does it compare to the state of the art (§5.3)?
- **RQ3**: How useful is it in real-world applications (§5.4)?

5.1 Experiment Setup

The evaluation is performed on the server with the Intel(R) Xeon(R) 8362 CPU at 2.80GHz, 256G memory, 1 Nvidia GeForce RTX 3080 Ti GPU for evaluating machine-learning-based baselines, and Ubuntu 22.04.

5.1.1 Datasets. We adopt nine real-world cryptographic libraries as objects, as listed in Table 2, with the lines of source code (LoC) and average numbers of cryptography-relevant functions in the compiled binaries. Those functions are captured by checking the keywords of forty known algorithms in function names, including AES, RSA, SM4, MD5, etc., in a case-insensitive manner [Hu et al. 2025]. These libraries are compiled with different compilers, including GCC v11.4.0, Clang v15.0.3, and ICX v2023.1.0, and diverse optimization options, i.e., O0, O1, O2, and O3. Then, the generated binaries are utilized for the comprehensive analysis of BinCrypto's accuracy, discernment, and efficiency. For the obfuscation experiment, without losing the generality, `libcrypto` is adopted as the target, because it serves the cryptographic component of `OpenSSL`, providing typical implementations of various standard algorithms. The binaries are obfuscated with the three widely used techniques provided by Obfuscator-LLVM (`OLLVM`) [Junod et al. 2015], including instruction substitution, bogus control flow, and control flow flattening.

Additionally, four common programs, `Curl`, `Nginx`, `OpenVPN`, and `PostgreSQL`, which employ cryptographic libraries to guarantee security, are utilized to check whether BinCrypto is capable of identifying statically-linked cryptographic functions. We also collect eight real-world samples from various ransomware families to further demonstrate the capability of BinCrypto in malware analysis, including `BlackSuit`, `DarkAngles`, `Erebus`, `LockBit`, `Monti`, `REvil`, `RansomEXX`, and `Royal`.

5.1.2 Ground Truth and Metrics. All the binaries for the evaluation are stripped. To verify the correctness of the results, for samples with source code, we compile their extra unstripped copies, adopting the symbols information as the ground truth. For others, we verify the results manually.

In our scenario, target functions are those under analysis, consisting of both cryptographic related as well as unrelated functions. We aim to evaluate the ability of BinCrypto to correctly identify the cryptographic-related functions among the target ones. BinCrypto compares each target function to all the reference cryptographic functions to calculate similarity scores in pairs. The match is considered to be correct if the reference, which scores the highest, shares the same name or implements the same algorithm with the target function. Then, we adopt **Recall@1** to evaluate the accuracy, which means the ratio of target cryptographic-related functions that achieve the correct match [Chandramohan et al. 2016; Feng et al. 2017; Marcelli et al. 2022; Wang and Wu 2017; Xu et al. 2023a,b]. The formula is as follows:

$$Recall@1 = \frac{|\{t \in T_c \mid n_t \in N_r\}|}{|T_c|}, \tag{1}$$

where T_c is the set of target cryptographic-related functions, n_t is the symbol name of the function t, and N_r is the symbol name set of the reference functions with the highest score.

Besides, since there might be samples with equal similarity scores, i.e., $|N_r| > 1$ in Equitation 1, we employ the **Single Match Ratio** to evaluate the discernment of a method, similar to the previous work [Zhou et al. 2024]. The formula is as follows:

$$Single\ Match\ Ratio = \frac{|\{t \in T_c \mid n_t \in N_r \wedge |N_r| = 1\}|}{|\{t \in T_c \mid n_t \in N_r\}|}. \tag{2}$$

5.1.3 Baseline Methods. We adopt WheresCrypto [Meijer et al. 2021], FindCrypt2 [Guilfanov 2006], Signsrch [Auriemma 2016], Yara4Ida [Weatherman 2022], jTrans [Wang et al. 2022], and Hermes [He et al. 2024b] as the baselines. WheresCrypto is the state-of-the-art method for binary cryptographic function identification. Provided with reference implementations, it achieves the goal by measuring the similarity of data flow graphs between the targets and references. FindCrypt2, Signsrch, and Yara4Ida are tools based on pre-defined heuristic patterns.

Table 3. Accuracy and discernment of BinCrypto. The three rows of the table header represent the configuration of the experiments, the compilation settings of the references, and that of the target functions, respectively. The two sub-columns of each pair of analysis list the values of Recall@1 and Single Match Ratio. We failed to compile libfreeblpriv3 with Clang-O3 and ICX-O3.

Library	Cross-optimization						Cross-compiler				Cross-both			
	GCC-O0		Clang-O0		ICX-O0		GCC-O3				GCC-O0			
	GCC-O3		Clang-O3		ICX-O3		Clang-O3		ICX-O3		Clang-O3		ICX-O3	
GnuTLS	.797	.679	.776	.720	.785	.727	.861	.673	.858	.672	.778	.695	.790	.700
cryptolib	.722	.877	.734	.940	.694	.956	.899	.873	.840	.973	.730	.912	.719	.899
Mbed-TLS	.669	.647	.664	.544	.624	.554	.762	.634	.661	.664	.666	.618	.600	.641
wolfSSL	.604	.762	.561	.759	.536	.783	.678	.763	.646	.772	.575	.730	.556	.741
Libgcrypt	.674	.723	.667	.684	.680	.697	.762	.812	.735	.797	.602	.777	.629	.795
Nettle	.818	.702	.870	.748	.837	.724	.750	.745	.687	.746	.721	.751	.705	.733
libfreeblpriv3	.636	.844	-	-	-	-	-	-	-	-	-	-	-	-
LibreSSL	.763	.681	.782	.672	.741	.688	.689	.715	.779	.717	.721	.686	.757	.688
libcrypto	.840	.733	.846	.734	.822	.715	.830	.724	.857	.742	.779	.719	.813	.733
Average	.742	.702	.753	.686	.726	.684	.766	.696	.766	.705	.712	.683	.704	.690

jTrans and Hermes are selected because they are generic and state-of-the-art machine-learning-based methods for function similarity analysis, which support code identification as well. They are based on the Transformer architecture [Vaswani et al. 2017] and GNN [Li et al. 2015], respectively. The two methods are trained and fine-tuned using their provided datasets, which contains the cryptographic binaries, e.g., OpenSSL. We use the baseline methods to complete the same analysis tasks for the comparison. Although WheresCrypto is designed for ARM architecture, its core technique, data flow graph isomorphism, is platform independent, as is BinCrypto. We then compile the libraries in Table 2 into ARM binaries for WheresCrypto in order to perform the comparison.

BinCrypto and the baselines above are all methods without actual execution. It is unfair to directly compare them to the dynamic methods, such as CryptoHunt [Xu et al. 2017b], which have their own inherent limitations. That would be discussed in Section 6.4.

5.2 RQ1: Performance

In this section, BinCrypto is evaluated to show its accuracy, discernment (§5.2.1), and efficiency (§5.2.2). Three sets of experiments are conducted with diverse configurations: i) *cross-optimization*, with reference and target functions compiled with different optimization levels but the same compiler; ii) *cross-compiler*, with the functions compiled with different compilers but the same optimization; and iii) *cross-both*, with both different compilers and optimization levels. The reference and target binaries are compiled from the same codebase.

5.2.1 Accuracy and Discernment. The results are listed in Table 3. For experiments with different optimizations, it only contains those between *O3* and *O0*, which has the most significant differences in binaries [Hu et al. 2021].

Overall, for cross-optimization analysis in the table, BinCrypto achieves *74.1%* of Recall@1 and *69.1%* of Single Match Ratio on average, and the values of cross-compiler analysis are *76.6%* and *70.0%*, indicating that optimizations pose more difficulties for the identification. When the results

Table 4. Results compared with existing work. The reference code is compiled with GCC-O0. The first sub-column of each method lists the values of Recall@1. The second sub-column (if it exists) lists those of Single Match Ratio.

Target	BinCrypto		WheresCrypto[1]		FindCrypt2		Signsrch		Yara4Ida		jTrans		Hermes	
GCC-O3	.745	.683	.487	.497	.079	1.000	.295	1.000	.392	1.000	.500	.552	.248	.992
Clang-O3	.712	.683	.342	.633	.086	1.000	.314	1.000	.411	1.000	.432	.508	.136	1.000
Clang-O0	.904	.709	.416	.519	.093	1.000	.287	1.000	.378	1.000	.620	.540	.535	1.000
Average	.796	.693	.409	.549	.087	1.000	.298	1.000	.393	1.000	.526	.535	.309	.998
Time / Function	2.817s		6.695s		< 0.001s		< 0.001s		< 0.001s		0.002s		0.003s	

[1] inline depth $d = 4$, the depth level to inline subroutines

of all the experimental configurations are involved, the average Recall@1 and Single Match Ratio are *84.5%* and *73.3%* for cross-optimization, *86.3%* and *74.0%* for cross-compiler, *71.4%* and *71.5%* for cross-both analysis, and *83.2%* and *73.3%* for all the sets of experiments [Hu et al. 2025].

By investigating the results, we find that the incorrectness primarily stems from two reasons. One is **function inlining**, which is a vital optimization adopted in modern compilers [Chen et al. 1993]. It replaces the call site in a caller with the callee to pursue better time performance. The code features collected by BinCrypto are fused accordingly. Different compilers also adopt diverse heuristics to decide whether to perform inlining [Theodoridis et al. 2022]. BinCrypto fails to infer if an invoked function, including library functions (e.g., `memcpy`), could be inlined in all cases, even though selective inlining [Chandramohan et al. 2016] is adopted. Then, it is highly possible for BinCrypto to miss the match, if one function has inlined code, whereas its counterpart does not.

The other is the **varying granularities of memory access**. Binary code accesses memory with multiple granularities, i.e., bytes, words, double-words, or quad-words. When processing a large amount of data, multi-byte accessing is more efficient than single-byte accessing. Therefore, cryptographic functions would be optimized to access data in multiple bytes at a time rather than a single byte. That leads to the mismatching, because BinCrypto fails to align features of different granularities.

5.2.2 Efficiency. Similarity measurement is the most time-consuming component of BinCrypto, which employs the LCS algorithm with the time complexity of $O(mn)$. Although it takes more time as the feature sequences become longer, BinCrypto processes each basic block only once, balancing analysis efficiency and accuracy/discernment. As a result, in the experiments, the analysis of `Nettle` is completed in the least amount of time, i.e., *105.756*s (*0.183*s / Function), and `LibreSSL` costs *11.31*h (*8.272*s / Function), the most amount of time in the experiments. On average, BinCrypto takes *3.081*s to process one function. Similarity measurement accounts for *95.3%* of the processing time, i.e., *2.936*s, while preprocessing and code emulation occupy *1.0%* and *3.7%* of the time separately.

Answer to RQ1: BinCrypto achieves the average Recall@1 of *83.2%* and Single Match Ratio of *73.3%* for all the experiments. The incorrectness is mainly caused by function inlining and varying granularities of memory access. It spends *3.081*s processing one function on average, where similarity measurement accounts for most of the time, i.e., *95.3%*.

5.3 RQ2: Comparison with Existing Work

In this section, BinCrypto is compared to the baseline work. The results are listed in Table 4, where the reference code is compiled with GCC-O0.

5.3.1 WheresCrypto. WheresCrypto is the state-of-the-art method depending on data flow graph isomorphism between the reference and target functions. It defines a parameter of inline depth (d), which denotes the depth of subroutines that are inlined [Meijer et al. 2021]. We choose the setting of $d = 4$ in this section, the same as that adopted in the original paper. BinCrypto achieves better performance from the perspectives of accuracy, discernment, and efficiency. The input and output features it adopts are more robust than data flow graph structures across code transformation caused by compiler optimization. Path-insensitive emulation along with LCS computation is also faster than graph isomorphism analysis which belongs to NP [Babai 2016].

5.3.2 FindCrypt2, Signsrch, and Yara4Ida. The three methods identify cryptographic functions with heuristic patterns. Specifically, FindCrypt2 relies on the specified constants of cryptographic algorithms, such as the S-boxes of symmetric ciphers, initialization vector values of hash functions, etc. Signsrch further attempts to search for keyword strings of those algorithms in binaries, based on which Yara4Ida optimizes the searching patterns to improve the performance. Their Recall@1 values are much lower than BinCrypto, because heuristics can only cover limited cases. Their values of Single Match Ratio are all 100.0% due to the nature of heuristics-based patterns which are stringent. Namely, when a pattern is matched, there is a high likelihood that it is correct.

5.3.3 jTrans and Hermes. jTrans is based on machine-learning techniques to generally detect binary similar functions. With the Transformer architecture, it attaches more attention to control flow information by embedding the source and destination of a jump instruction [Wang et al. 2022]. The model is fine-tuned with its official dataset `BinaryCorp` [Wang et al. 2024b] which contains binaries of cryptographic libraries, i.e.,`OpenSSL`. Hermes adopts GGNN [Li et al. 2015] to embed the control flow graph along with data dependence information and function calling information [He et al. 2024b]. By default, it is trained with the dataset proposed by Marcelli et al. [Marcelli et al. 2022], which also contains `OpenSSL`. The test datasets in our evaluation are composed of cryptographic functions, which involves numerous arithmetic and logic operations. The long-range relationships between code and variables are common as well. Thus, jTrans achieves higher average Recall@1 than Hermes in this section. BinCrypto outperforms jTrans and Hermes with better accuracy, because it depends on the behavior information of cryptographic functions rather than the code graph properties, which are prone to compiler code transformation.

Answer to RQ2: BinCrypto achieves better Single Match Ratio than WheresCrypto and jTrans and higher Recall@1 than all the baseline methods, because it is more resilient to code transformation. It is more efficient than WheresCrypto as well due to path-insensitive emulation.

5.4 RQ3: Applications

In this section, we show the potential practical applications of BinCrypto in the following scenarios: i) *statically-linked library analysis*, to detect cryptographic functions statically linked in real-world programs; ii) *cross-library analysis*, to identify target functions with references of the same algorithm but compiled from different codebases; iii) *obfuscated code analysis*, to analyze binaries with code obfuscation; iv) *malware analysis*, to locate the cryptographic functions in a ransomware.

5.4.1 Statically-Linked Library Analysis. Cryptographic libraries are usually statically linked into other programs to provide encryption and authentication services. It is meaningful to locate the cryptographic functions in those programs in order to facilitate the downstream security analysis, e.g., patch analysis [Zhang and Qian 2018]. `libcrypto` is selected as the reference, because it

Table 5. Results of statically-linked library analysis. The references are from `libcrypto`.

Target Program	# of Functions	Recall@1	Total Time (min)
Curl	7,529	31 / 34	9.792
Nginx	8,042	31 / 34	11.135
OpenVPN	9,953	31 / 34	15.428
PostgreSQL	30,541	31 / 34	999.414

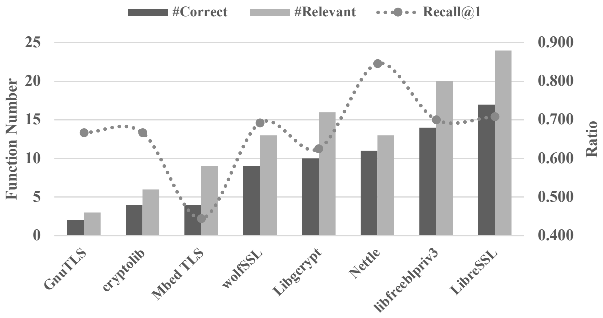


Fig. 8. Results of cross library analysis. The references are from `libcrypto`.

constitutes the cryptographic component of `OpenSSL`, which is widely utilized in practice. We then use BinCrypto to find the cryptographic functions in the real-world target programs.

The results are presented in Table 5. The second column lists the number of functions contained in the target program. The third column shows the values of Recall@1. Note that, in the experiments, we concentrate on the *34* core functions that implement *19* standard cryptographic algorithms [Hu et al. 2025], including AES, MD5, RSA, etc. The last column provides the total processing time in minutes.

Overall, BinCrypto achieves high Recall@1. *31* of the *34* functions are successfully identified for each program, and *17* have a single correct match. The three incorrect samples are the same for the four programs, i.e., *rsa_ossl_private_encrypt*, *rsa_ossl_private_decrypt*, and *sha3_update*, which are caused by function inlining (§5.2.1). It takes more than *15* hours to process `PostgreSQL` which has more than thirty thousand functions, and the average time for each function is *1.963*s.

5.4.2 Cross-Library Analysis. Cryptographic algorithms could have a variety of implementations. Typically, the nine libraries in Table 2 provide diverse implementations for the standard algorithms. Thus, in this section, we show the capability of BinCrypto to identify target functions which implement the same algorithm but are compiled from different codebases.

Similar to Section 5.4.1, `libcrypto` is selected as the reference to identify the target functions of the other eight libraries in Table 2, and we only focus on the *34* core cryptographic functions [Hu et al. 2025]. Since function symbol names vary across libraries, we verify the correctness of the analysis manually. The results are depicted in Figure 8. Except for `Mbed-TLS`, whose Recall@1 is *44.4%*, the values of others are all over *60.0%*, and the average is *68.3%*. Besides, `GnuTLS` mainly adopts `Nettle` as the external library to implement cryptographic operations. Its target function number then is small in the experiments.

Table 6. Recall@1 of analyzing binaries obfuscated by OLLVM. The targets are from libcrypto.

Obfuscation	BinCrypto	WheresCrypto[1]	jTrans	Yara4Ida	Hermes
SUB	.541	.064	.337	.285	.351
BCF	.595	.239	.320	.194	.114
FLA	.513	.119	.310	.143	.231
Average	.550	.141	.323	.204	.224

[1] inline depth $d = 4$, the depth level to inline subroutines

In addition to the aforementioned reasons (§5.2.1), **variant implementations** cause the incorrectness of cross-library analysis as well. For example, there are more indirect calls in the binary code of the reference library libcrypto than in other libraries, leading to mismatching because BinCrypto cannot resolve the invalid emulated target of indirect calls. On the other hand, libcrypto processes the Rijndael T-tables [Chen 2020; Daemen and Rijmen 1999] in a little-endian manner, while cryptolib does it in a big-endian manner. As a result, the code features collected are also in different orders, which lowers the similarity score.

5.4.3 *Obfuscated Code Analysis.* In this section, we show the capability of BinCrypto in analyzing binaries with obfuscation. The code is obfuscated by Obfuscator-LLVM (OLLVM) in the experiments. The results are presented in Table 6 with those of baseline methods attached as well. For the heuristics-based tools, we only consider Yara4Ida in this section as the reference, because it achieves better performance than FindCrypt2 and Signsrch in the previous experiments (§5.3, Table 4). For the machine-learning-based methods, it is a common practice to train them with benign code and test them with obfuscated code [Ding et al. 2019]. In this section, we adopt jTrans and Hermes, which have been trained with their default datasets. To clarify, without sacrificing generality, all functions in target binaries are obfuscated with the default settings of OLLVM to achieve a comprehensive comparison between different methods. As a result, due to the complete code coverage for extracting behavior-related features, BinCrypto attains an average Recall@1 of *55.0%*, outperforming the baselines. The baselines are vulnerable to code obfuscation, because of their reliance on the code graph properties, e.g., data flow graphs, control flow information, data dependence relationships, etc.

OLLVM adopts three widely used obfuscation techniques. Instruction substitution (SUB) replaces original operators with equivalent but more complicated instructions, confusing the data flow relationships among variables. Hence, WheresCrypto, which relies on the isomorphism of data flow graphs, demonstrates the lowest Recall@1 when dealing with SUB (*6.4%*). Bogus control flow (BCF) adds opaque predicates to a basic block to break it into two. It creates redundant paths and complicates control flow graphs. As a result, BCF poses greater threats to jTrans and Hermes than SUB does, with the resulting Recall@1 of 19.4% and 11.4% respectively, because of their reliance on the control flow information. Control flow flattening (FLA) breaks up functions into code blocks and stitches them via a dispatcher structure, e.g., the switch structure. It also complicates the data/control flow graphs and program dependence relationships. That could be indicated by the Recall@1 of WheresCrypto, jTrans, and Hermes, which is 11.9%, 14.3%, and 23.1%, separately.

BinCrypto is a static-analysis-based method without actual code execution. The effects of code obfuscation on it are also unignorable. SUB creates extra I/O operations which are irrelevant to the original semantics. The redundant paths generated by BCF lead to noisy code features as well. FLA leverages indirect jumps to implement the selective structures, which pose obstacles for the emulation to collect code features (§3.3.3). In practice, to maximize the effectiveness of BinCrypto,

Table 7. Results of cryptographic function identification in ransomware. ✓ means found and ✗ is not found.

Ransomware	Algorithm	BinCrypto	Yara4Ida	jTrans	Hermes
BlackSuit	AES	✓	✓	✓	✗
DarkAngles	AES	✓	✓	✗	✗
Erebus	RC4	✓	✗	✓	✓
	RSA	✓	✗	✓	✗
LockBit	Blake2b	✓	✗	✓	✗
	ECC	✓	✗	✗	✓
Monti	AES	✓	✓	✗	✗
REvil	AES	✓	✓	✗	✗
RansomEXX	AES	✓	✗	✗	✓
Royal	AES	✓	✓	✓	✓

when faced with obfuscated binaries, it would be a better choice to deobfuscate them first for further analysis [Schloegel et al. 2022; Xu et al. 2018].

5.4.4 Malware Analysis. Malware analysis is one of the typical applications of binary cryptographic function identification. We then evaluate BinCrypto with real-world ransomware. Among the eight samples we collect, six of them rely on AES to perform the encryption, including `BlackSuit`, `DarkAngles`, `Monti`, `REvil`, `RansomEXX`, and `Royal`. `Erebus` employs RC4 to encrypt files and the RC4 key is then encrypted by RSA. `LockBit` adopts a hybrid-cryptography scheme for the encryption, including Blake2b and ECC.

The results are listed in Table 7. BinCrypto successfully identifies the target cryptographic functions in the collected ransomware, due to the complete code coverage and the reliance on the behavior-related code features. By contrast, Yara4Ida is a tool based on heuristic patterns. It fails to recognize cryptographic implementations when lacking dependable static patterns. For example, it is able to find AES with S-box in the `.rodata` section. It fails to locate AES in `RansomEXX`, because the S-box is generated dynamically [Singh et al. 2017] instead of stored in the data section statically. jTrans depends on the control flow information embedded by the Transformer architecture for code similarity analysis. Hermes employs GNN to embed program dependence relationships as features in addition to the control flow information. They are unable to accurately locate all the cryptographic algorithms in the target ransomware, as the features they adopt are susceptible to code transformation, e.g., optimization and obfuscation.

Answer to RQ3: BinCrypto shows its potential for practical applications, including statically-linked library analysis, cross-library analysis, obfuscated code analysis, and malware analysis.

6 Discussion and Future Work

6.1 Application Scope

The prototype of BinCrypto is implemented to handle 64-bit ELF files on Linux, while the proposed method is platform-independent. The experimental results demonstrate its effectiveness and potential applications. With some engineering work, it would be able to handle PE files on Windows and perform analysis across different instruction set architectures, e.g., ARM, MIPS, etc. The adoption of `QEMU` indicates such capability as well.

6.2 Obfuscation

Although BinCrypto is shown to be capable of analyzing obfuscated code generated by `OLLVM` (§5.4.3), that does not mean it could handle all kinds of obfuscation. Additionally, BinCrypto performs much better when processing unobfuscated code (§5.2). Since deobufucation is a well-established research field [Blazytko et al. 2017; David et al. 2020; Schloegel et al. 2022; Xu et al. 2018; Yadegari et al. 2015], in practice, it is recommended to deobfuscate the obfuscated code first for further analysis so as to maximize the effectiveness of BinCrypto.

6.3 Function Inlining

Function inlining is the main reason causing the incorrectness (§5.2.1). Selective inlining [Chandramohan et al. 2016] mitigates the problem but does not resolve it. Since the inlining decision is made mainly based on heuristics [Theodoridis et al. 2022], machine-learning-based solutions might be effective in inferring whether a callee could be inlined, which is left as future work.

6.4 Comparison with Dynamic Methods

BinCrypto is based on static analysis; it proactively reasons about program behaviors using static analysis techniques to infer ranges of variable values in a path-insensitive manner. QEMU is employed as its underlying abstract interpretation engine for the implmentation. This is a key distinction from dynamic analysis, which relies on concrete executions with various inputs, hoping to satisfy path constraints by chance. Although the dynamic methods are more resilient to code obfuscation, they have their own inherent limitations, e.g., code coverage and overhead. It is difficult to generate input which covers arbitrary code. Taking malware analysis as an example, if the malicious behaviors or the cryptographic functions only constitute a small portion of a host program whose primary functionality is benign, static methods might be more suitable for such cases. Hence, static and dynamic methods both have their own advantages, which could complement each other for practical applications.

On the other hand, due to the usage of static techniques, BinCrypto is able to achieve more comprehensive and efficient analysis results. Its efficiency significantly surpasses existing dynamic methods. On average, BinCrypto analyzes a cryptographic function in just a few seconds. In contrast, the online trace logging of CryptoHunt [Xu et al. 2017b], a state-of-the-art dynamic binary cryptographic function identification tool, causes 5-6X slowdown. Furthermore, its offline analysis takes tens of minutes to locate one target cryptographic function in a binary.

6.5 Proprietary Cryptographic Function Identification

In the literature, cryptographic function identification generally requires oracles, i.e., the reference implementations, and so does BinCrypto. WheresCrypto loosens the requirement, handling proprietary cryptographic primitives by recognizing known structures, e.g., Feistel structure, which unfortunately still suffers from binary code transformation. It tends to underperform in analyzing even standard cryptographic functions in our experiments (§5.3), which motivates BinCrypto. Understanding binary proprietary cryptographic algorithm is essential for software security analysis. It requires to figure out how the key is scheduled and how the plaintext is processed to generate the ciphertext. We will leave it as future work.

7 Related Work

7.1 Binary Cryptographic Function Detection

The heuristics-based methods rely on the presence of loops, entropy, the ratio of logic operations, and the avalanche effect [Ramanujam and Karuppiah 2011] to infer if functions are cryptography relevant [Auriemma 2016; Guilfanov 2006; Li et al. 2012; Wang et al. 2009], which cannot cover all the cases in the real world. Graph-isomorphism-based methods adopt data flow graphs as the signature to achieve the goal [Lestringant et al. 2015; Meijer et al. 2021]. However, they are still prone to code transformation of binaries, e.g., compiler optimization. The Input/Output-relation-based methods then collect input and output values during execution for the identification [Calvet et al. 2012; Gröbert et al. 2011]. Additionally, CryptoHunt [Xu et al. 2017b] depends on the runtime values of loops. Unfortunately, these methods depend on dynamic analysis, which are insufficient for practical use because of the inherent limitations, such as coverage and efficiency issues. By contrast, BinCrypto relies on code behavior information which is captured via emulation in a path-insensitive manner, balancing analysis accuracy, efficiency, and code coverage.

7.2 Binary Similarity Analysis

Binary similarity analysis has a lot of important applications in software security. Syntax- and structure-based methods adopt normalized opcode and operand sequences, control flow graphs (CFGs), and call graphs as features to measure code similarity [David and Yahav 2014; Ding et al. 2016; Flake 2004; Sæbjørnsen et al. 2009], which cannot handle code compiled with different compilation configurations. Symbolic-execution-based methods perform equivalence checking between code constrains to achieve the goal [Luo et al. 2014, 2017; Zhang et al. 2014], which suffer from the efficiency issue of solvers. Emulation-based methods emulate code execution and measure the similarity based on the emulated values [Chandramohan et al. 2016; Eschweiler et al. 2016; Hu et al. 2017; Pewny et al. 2015; Wang and Wu 2017; Xu et al. 2023b; Xue et al. 2018]. The mainstream emulation and execution-based methods have limitations in terms of code coverage except BLEX [Egele et al. 2014]. BLEX achieves complete code coverage by executing the target code repeatedly, starting each time from the uncovered instruction, until all the code is covered at least once. To avoid state explosion (Figure 2b), it executes uncovered code with brand new program state, which however disregards the context of execution. It potentially runs equivalent code with different inputs, which diminishes its analysis accuracy. IMF-sim [Wang and Wu 2017] and BinMatch [Hu et al. 2021] are dynamic methods with limited code coverage, which undermines their practicability. ARCTURUS [Zhou et al. 2024] traverses the code under the guidance of reachability, whose effectiveness is proven on the assumption that two pieces of code for comparison are compiled from the same codebase. By contrast, because of interval-based path-insensitive code emulation, BinCrypto would be more practical for achieving complete code coverage efficiently. Besides, BinCrypto is capable of analyzing code implementing the same algorithm but generated from different codebases (§5.4.2). It is more aligned with equivalent algorithm analysis than similar code analysis. More recently, machine learning techniques have also been widely utilized for binary similarity analysis [Ding et al. 2019; Feng et al. 2016; He et al. 2024b,a; Jiang et al. 2024; Li et al. 2024; Liu et al. 2018; Luo et al. 2023; Pei et al. 2022; Wang et al. 2024a, 2022; Xu et al. 2017a; Yang et al. 2021; Yu et al. 2020a,b; Zuo et al. 2019]. These methods are robust when the training set is well constructed, while it remains an open question to extract and embed the precise semantics of binary code statically [Marcelli et al. 2022; Zhang et al. 2023].

8 Conclusion

We propose BinCrypto to identify cryptographic functions in binaries. It extracts inputs and outputs as code features via code emulation, and performs path-insensitive analysis to achieve complete basic block coverage efficiently. The evaluation shows that BinCrypto outperforms the state-of-the-art methods from the perspective of accuracy and discernment. The experiments also demonstrate its potential for practical applications.

Data-Availability Statement

The artifact is available on github (https://github.com/yikunh/BinCrypto) and Zenodo [Hu et al. 2025]. To ensure flexibility, it is provided as a Docker image based on Linux, containing the sample binaries and compiled executables of BinCrypto's prototype. The samples are presented unstripped to enhance the readability of the results, while BinCrypto **does not** rely on the symbol and debug information in any way. It demonstrates how the method emulates the samples to extract code features, perform the comparison, and present the results.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. The authors would also like to extend special thanks to Xiangzhe Xu from Purdue University for cross-validating the baseline method. Yituo He and Yizhe Cui successively helped implement and maintain the prototype. The SJTU authors were partially supported by the National Natural Science Foundation of China (No. U2336210), Shanghai Pujiang Program (No. 22PJ1405700), and Shanghai Committee of Science and Technology, China (No.23511101000). The HKUST author is supported by an NSFC/RGC JRS grant under the contract N_HKUST605/23.

References

Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. 2000. Specification of Camellia-a 128-bit block cipher. *Specification Version* 2 (2000). doi:10.25148/etd.fi14051800

C Ashokkumar, Bholanath Roy, M Bhargav Sri Venkatesh, and Bernard L Menezes. 2018. " S-Box" Implementation of AES is NOT side-channel resistant. *Cryptology ePrint Archive* (2018). doi:10.1007/s41635-019-00082-w

Luigi Auriemma. 2016. Signsrch 0.2.4. http://aluigi.altervista.org/mytoolz.htm. Tool searches encryption/compression algorithms inside files.

László Babai. 2016. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 684–697. doi:10.59350/vre69-edk82

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. California, USA, 46. doi:10.1109/csma.2015.14

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*. Springer, 193–207. doi:10.21236/ada360973

Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the semantics of obfuscated code. In *26th USENIX Security Symposium (USENIX Security 17)*. 643–659. doi:10.1109/sere.2012.13

Joan Calvet, José M Fernandez, and Jean-Yves Marion. 2012. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 169–182. doi:10.1145/2382196.2382217

Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689. doi:10.1145/2950290.2950350

William Y Chen, Pohua P. Chang, Thomas M Conte, and Wen-mei W. Hwu. 1993. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput.* 42, 9 (1993), 1045–1057. doi:10.1109/12.241594

Xiaoqi Chen. 2020. Implementing AES encryption on programmable switches via scrambled lookup tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*. 8–14. doi:10.1145/3405669.3405819

Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 515–530. doi:10.1145/3319535.3354249

Irving M Copi, Carl Cohen, and Kenneth McMahon. 2016. *Introduction to logic*. Routledge. doi:10.4324/9780203204887_chapter_10

Joan Daemen and Vincent Rijmen. 1999. AES proposal: Rijndael. (1999). doi:10.1007/springerreference_461

S Das, JKMS Uz Zaman, and RJPT Ghosh. 2013. Generation of AES S-Boxes with various modulus and additive constant polynomials and testing their randomization. *Procedia Technology* 10 (2013), 957–962. doi:10.1016/j.protcy.2013.12.443

Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. Qsynth-a program synthesis based approach for binary code deobfuscation. In *BAR 2020 Workshop*. doi:10.14722/bar.2020.23009

Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360. doi:10.1145/2594291.2594343

Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2016. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 461–470. doi:10.1145/2939672.2939719

Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489. doi:10.1109/sp.2019.00003

Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*. 303–317. doi:10.2172/5154317

Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *Ndss*, Vol. 52. 58–79. doi:10.14722/ndss.2016.23185

Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 346–359. doi:10.1145/3052973.3052995

Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 480–491. doi:10.1145/2976749.2978370

Halvar Flake. 2004. Structural comparison of executable objects. *DIMVA 2004, July 6-7, Dortmund, Germany* (2004). doi:10.1049/cp.2013.2196

Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. 1075–1092. doi:10.1007/springerreference_63978

Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated identification of cryptographic primitives in binary programs. In *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*. Springer, 41–60. doi:10.1007/978-3-642-23644-0_3

Ilfak Guilfanov. 2006. FindCrypt2. https://hex-rays.com/blog/findcrypt2/. IDA Pro plug-in searches for cryptographic algorithm.

Jongbeen Han, Mansub Song, Hyeonsang Eom, and Yongseok Son. 2021. An Efficient Multi-Signature Wallet in Blockchain Using Bloom Filter. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC'21)*. Association for Computing Machinery, New York, NY, USA, 273–281. doi:10.1145/3412841.3441910

Paul Havlak. 1997. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 4 (1997), 557–567. doi:10.1145/262004.262005

Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024b. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*. doi:10.14722/bar.2024.23006

Kaiyan He, Yikun Hu, Xuehui Li, Yunhao Song, Yubo Zhao, and Dawu Gu. 2024a. Strtune: Data Dependence-Based Code Slicing for Binary Similarity Detection With Fine-Tuned Representation. *IEEE Transactions on Information Forensics and Security* 19 (2024), 10233–10245. doi:10.1109/tifs.2024.3484944

Hex-rays. 2024. https://hex-rays.com/ida-pro/. A disassembler for computer software which generates assembly language source code from machine-executable code.

Sabine Houy, Philipp Schmid, and Alexandre Bartel. 2023. Security Aspects of Cryptocurrency Wallets—A Systematic Literature Review. *ACM Comput. Surv.* 56, 1, Article 4 (aug 2023), 31 pages. doi:10.1145/3596906

Yikun Hu, Yituo He, Wenyu He, Haoran Li, Yubo Zhao, Shuai Wang, and Dawu Gu. 2025. Binary Cryptographic Function Identification via Similarity Analysis with Path-insensitive Emulation. doi:10.5281/zenodo.14943895

Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. 2021. A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison. *IEEE Transactions on Software Engineering* 47, 6 (June 2021), 1241–1258. doi:10.1145/2635868.2635900

Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 88–98. doi:10.1109/icpc.2017.22

Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639100

Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May*

19th, 2015, Brecht Wyseur (Ed.). IEEE, 3–9. doi:10.1109/spro.2015.10

Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 2015. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 203–214. doi:10.1145/2714576.2714639

Weilong Li, Jintian Lu, Ruizhi Xiao, Pengfei Shao, and Shuyuan Jin. 2024. RCFG2Vec: Considering Long-Distance Dependency for Binary Code Similarity Detection. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 770–782. doi:10.1145/3691620.3695070

Xin Li, Xinyuan Wang, and Wentao Chang. 2012. CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution. *IEEE transactions on dependable and secure computing* 11, 2 (2012), 101–114. doi:10.1109/tdsc.2012.83

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015). doi:10.21203/rs.3.rs-1364332/v1

Yan Lin and Debin Gao. 2021. When function signature recovery meets compiler optimization. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–52. doi:10.1109/sp40001.2021.00006

Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. *α*diff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 667–678. doi:10.1145/3238147.3238199

Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. 2023. Decompiling x86 deep neural network executables. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7357–7374. doi:10.21236/ada449077

HJ Lu, Michael Matz, J Hubicka, A Jaeger, and M Mitchell. 2018. System V application binary interface. *AMD64 Architecture Processor Supplement* (2018), 588–601. doi:10.3403/00374100u

Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 389–400. doi:10.1145/2635868.2635900

Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. doi:10.1145/2635868.2635900

Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.. In *NDSS*. doi:10.14722/ndss.2023.24415

Noé Lutz. 2008. Towards revealing attacker's intent by automatically decrypting network traffic. *Mémoire de maıtrise, ETH Zürich, Switzerland* (2008). doi:10.1016/s1353-4858(19)30098-4

Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*. 2099–2116. doi:10.1142/9789811224317_0003

Carlo Meijer, Veelasha Moonsamy, and Jos Wetzels. 2021. Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code. In *30th USENIX Security Symposium (USENIX Security 21)*. 555–572. doi:10.1145/2714576.2714639

Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1187–1198. doi:10.1016/s0007-8506(07)63015-7

Niels Möller. 2013. Nettle. https://www.lysator.liu.se/~nisse/nettle. A low-level cryptographic library.

Mozilla. [n. d.]. Network Security Services (NSS). https://firefox-source-docs.mozilla.org/security/nss/index.html.

Gavin O'Gorman and Geoff McDonald. 2012. *Ransomware: A growing menace.* Symantec Corporation Arizona, AZ, USA. doi:10.1108/oxan-es272348

Rolf Oppliger. 2009. *SSL and TLS: Theory and Practice.* Artech House, Inc., USA. doi:10.7838/jsebs.2017.22.2.169

Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering* (2022). doi:10.1109/tse.2022.3231621

Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 709–724. doi:10.1109/sp.2015.49

POODLE. 2014. CVE-2014-3566. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3566. Accessed: 2024-01.

NGUYEN Anh Quynh and DANG Hoang Vu. 2015. Unicorn: Next generation cpu emulator framework. *BlackHat USA* 476 (2015). doi:10.1109/hcs49909.2020.9220443

Sriram Ramanujam and Marimuthu Karuppiah. 2011. Designing an algorithm with high Avalanche Effect. *IJCSNS International Journal of Computer Science and Network Security* 11, 1 (2011), 106–111. doi:10.21203/rs.3.rs-4113962/v1

Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 117–128. doi:10.1145/1572272.1572287

Nitin Saxena. 2009. Progress on Polynomial Identity Testing. *Bull. EATCS* 99 (2009), 49–79. doi:10.1007/978-3-319-05446-9_7

Nitin Saxena. 2014. Progress on polynomial identity testing-II. *Perspectives in Computational Complexity: The Somenath Biswas Anniversary Volume* (2014), 131–146. doi:10.1007/978-3-319-05446-9_7

Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 3055–3073. doi:10.1109/spro.2015.16

Jacob T Schwartz. 1980. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)* 27, 4 (1980), 701–717. doi:10.1145/322217.322225

Amandeep Singh, Praveen Agarwal, and Mehar Chand. 2017. Analysis of development of dynamic s-box generation. *Comput. Sci. Inf. Technol* 5, 5 (2017), 154–163. doi:10.13189/csit.2017.050502

Vugranam C Sreedhar, Guang R Gao, and Yong-Fong Lee. 1996. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 6 (1996), 649–658. doi:10.1145/236114.236115

Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 977–989. doi:10.1145/3503222.3507744

Sebastian Unger and Frank Mueller. 2002. Handling irreducible loops: optimized node splitting versus DJ-graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 4 (2002), 299–333. doi:10.1145/567097.567098

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017). doi:10.5040/9781350101272.00000005

Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)* 21, 1 (1974), 168–173. doi:10.1016/0020-0190(90)90109-b

Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024a. CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 149–161. doi:10.1145/3650212.3652117

Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. sem2vec: Semantics-aware Assembly Tracelet Embedding. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–34. doi:10.1145/3569933

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13. doi:10.1145/3533767.3534367

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2024b. BinaryCorp. https://github.com/vul337/jTrans. a dataset for the task of binary code similarity detection.

Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*. doi:10.14722/ndss.2017.23225

Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*. 627–642. doi:10.14722/bar.2019.23058

Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330. doi:10.1109/ase.2017.8115645

Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. 2009. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *Computer Security – ESORICS 2009*, Michael Backes and Peng Ning (Eds.). Vol. 5789. Springer Berlin Heidelberg, Berlin, Heidelberg, 200–215.

Kevin Weatherman. 2022. Yara for IDA 1.1.0. https://github.com/kweatherman/yara4ida. IDA Pro plugin with crypto/hash/-compression signatures.

Douglas Brent West et al. 2001. *Introduction to graph theory*. Vol. 2. Prentice hall Upper Saddle River. doi:10.1142/9789811273117_0001

Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., USA. doi:10.1109/hpca.2008.4658658

Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 442–458. doi:10.1145/3243734.3243827

Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017b. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 921–937. doi:10.1109/sp.2017.56

Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023a. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1106–1118. doi:10.1145/3597926.3598121

Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017a. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 363–376. doi:10.1109/eurosp53844.2022.00012

Xiangzhe Xu, Zhou Xuan, Shiwei Feng, Siyuan Cheng, Yapeng Ye, Qingkai Shi, Guanhong Tao, Le Yu, Zhuo Zhang, and Xiangyu Zhang. 2023b. PEM: Representing Binary Program Semantics for Similarity Analysis via a Probabilistic Execution Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 401–412. doi:10.1145/3611643.3616301

Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering* 45, 11 (2018), 1125–1149. doi:10.1109/tse.2021.3069529

Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 674–691. doi:10.1109/sp.2015.47

Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2224–2244. doi:10.1109/tse.2021.3056139

Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020a. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1145–1152. doi:10.1609/aaai.v34i01.5466

Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020b. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883. doi:10.31274/td-20240329-185

Fangfang Zhang, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Program logic based software plagiarism detection. In *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 66–77. doi:10.1109/issre.2014.18

Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*. 887–902. doi:10.1016/j.comcom.2021.03.011

Zhuo Zhang, Guanhong Tao, Guangyu Shen, Shengwei An, Qiuling Xu, Yingqi Liu, Yapeng Ye, Yaoxuan Wu, and Xiangyu Zhang. 2023. PELICAN: Exploiting Backdoors of Naturally Trained Deep Learning Models In Binary Code Analysis. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2365–2382. doi:10.54499/2020.09139.bd

Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 813–832. doi:10.1109/sp40001.2021.00051

Anshunkang Zhou, Yikun Hu, Xiangzhe Xu, and Charles Zhang. 2024. ARCTURUS: Full Coverage Binary Similarity Analysis with Reachability-guided Emulation. *ACM Transactions on Software Engineering and Methodology* 33, 4 (May 2024), 1–31. doi:10.1145/3640337

Richard Zippel. 1979. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*. Springer, 216–226. doi:10.1007/3-540-09519-5_73

Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. doi:10.14722/ndss.2019.23492

Received 2024-10-13; accepted 2025-02-18