SIGMADIFF: Semantics-Aware Deep Graph Matching for Pseudocode Diffing

Lian Gao*, Yu Qu*, Sheng Yu*†, Yue Duan‡ and Heng Yin*†
*University of California, Riverside
†Deepbits Technology Inc.
‡Singapore Management University
*{lgao027, yuq, syu061}@ucr.edu, *heng@cs.ucr.edu, ‡yueduan@smu.edu.sg

***Abstract*—Pseudocode diffing precisely locates similar parts and captures differences between the decompiled pseudocode of two given binaries. It is particularly useful in many security scenarios such as code plagiarism detection, lineage analysis, patch, vulnerability analysis, etc. However, existing pseudocode diffing and binary diffing tools suffer from low accuracy and poor scalability, since they either rely on manually-designed heuristics (e.g., Diaphora) or heavy computations like matrix factorization (e.g., DeepBinDiff). To address the limitations, in this paper, we propose a semantics-aware, deep neural network-based model called SIGMADIFF. SIGMADIFF first constructs IR (Intermediate Representation) level interprocedural program dependency graphs (IPDGs). Then it uses a lightweight symbolic analysis to extract initial node features and locate training nodes for the neural network model. SIGMADIFF then leverages the state-of-the-art graph matching model called Deep Graph Matching Consensus (DGMC) to match the nodes in IPDGs. SIGMADIFF also introduces several important updates to the design of DGMC such as the pre-training and fine-tuning schema. Experimental results show that SIGMADIFF significantly outperforms the state-of-the-art heuristic-based and deep learning-based techniques in terms of both accuracy and efficiency. It is able to precisely pinpoint eight vulnerabilities in a widely-used video conferencing application.**

I. INTRODUCTION

Pseudocode diffing, as a special kind of binary code diffing [9], [22], [38], [84], is a technique that precisely locates similar pseudocode tokens and captures the differences between the pseudocode (a.k.a., decompiled code) of two given binaries. It can play an essential role in many security scenarios such as code plagiarism detection [60], [68], lineage analysis [62], vulnerability and patch analysis [77].

In contrast to binary diffing that works at either function level or basic block level, pseudocode diffing provides a more fine-grained, semantic-rich, and human-comprehensible pseudocode token level similarity analysis. Hence, mainstream disassemblers such as IDA Pro [14] and Ghidra [23] can present binaries in pseudocode.

Compared to binary diffing, pseudocode diffing has several advantages. First, pseudocode, which is closer to higher-level languages like C than assembly, carries more semantic-level information. Therefore, it is more concise and human-readable. Second, pseudocode diffing compares two pseudocode programs at the token level, and thus is more fine-grained than binary diffing, which is at the basic block or function level. Third, it can achieve better accuracy than binary diffing in general since the semantic information recovered during decompilation (e.g., def-use relation and type information) can be very helpful during analysis. Fourth, pseudocode naturally supports cross-architecture diffing, as binaries on different architectures are all translated into one C-like language via decompilation.

However, pseudocode diffing brings unique challenges. Since much information is unavailable in stripped binaries (e.g., symbol information), decompilers have to infer the high-level features (e.g., variable names, expressions, high-level control constructs) for pseudocode and introduce a considerable amount of noise. That is why Diaphora [9] (the only pseudocode diffing tool to our knowledge) does not work well since it relies on a set of heuristics and performs simple string-based matching at the token level.

While the problem of pseudocode diffing evidently needs more study, plenty of research has been done on binary diffing. Traditional approaches like BinDiff [22] rely on heuristics (e.g., function name) to find similar functions and perform basic block matching along the control flow graph. These syntax-based heuristics are not robust and can be easily thwarted by compiler optimizations. Dynamic analysis-based approaches [39], [61], [75] are good at capturing the semantics of binaries and have good resilience against code obfuscation, but they struggle to cover much code, especially for large binaries. Learning-based approaches [37], [44], [76], [84] leverage machine learning techniques to encode graph information into numerical vectors, a.k.a, graph embeddings, and perform binary diffing. These techniques can distill unique semantic-level features of a program while avoiding heavy graph matching. Despite the reasonable accuracy and scalability, the majority of them only handle diffing at a coarse-grained function level. DeepBinDiff [38], the only basic block level binary diffing technique in this category, leverages the TADW algorithm [78] to generate context- and semantic-aware basic block embeddings and performs diffing. However, as shown in our evaluation, it does not scale well on large binaries and cannot be easily accelerated by GPUs due to the iterative algorithm in TADW.

Besides binary diffing, there also exists a line of research

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23208
www.ndss-symposium.org

for source code matching. GumTree [41] parses source code to abstract syntax tree (AST), and finds a sequence of "edit actions" in AST levels. ClDiff [52] improves previous approaches by grouping fine-grained code differences and summarizing high-level code changes. Nevertheless, the AST of pseudocode can change drastically due to code transformations introduced by compilers and/or decompilers. Consequently, AST-based techniques are not suitable for pseudocode diffing. Source code clone/similarity detection approaches [65], [80], [83] are either too coarse-grained or only consider syntax-level features and do not work well on pseudocode diffing, according to our evaluation.

In this paper, we present a novel pseudocode diffing technique, called SIGMADIFF[1]. To mitigate the challenges caused by syntax level changes and achieve better accuracy, we design SIGMADIFF based on two important observations: 1) a large number of syntax level changes in decompilation are introduced during the transformation from intermediate representation (IR) to pseudocode; 2) the dependency information (e.g., control and data dependencies) among IRs can be of great help for matching and diffing.

To this end, we first perform diffing at IR level and map the IR-level diffing results up to the pseudocode level. We construct interprocedural program dependency graphs (IPDGs) [45] and extract a symbolic expression for each IR to capture the semantic meanings of a binary. After that, we model the program-wide IR diffing problem as a graph matching problem on two IPDGs from the two given binaries by leveraging the deep graph matching consensus (DGMC) model [46] to fully exploit the neighboring contextual information.

We have implemented a prototype of SIGMADIFF, and conducted extensive experiments to evaluate its efficacy. Experimental results show that SIGMADIFF outperforms Diaphora in cross-version, cross-optimization-level, cross-compiler, and cross-architecture diffing tasks. For instance, it outperforms Diaphora by 308%, 85%, 38% in terms of F1-scores in O0 vs. O3, O1 vs. O3, and O2 vs. O3, respectively. We also compare with DeepBinDiff by mapping its basic block level results to pseudocode tokens and show that SIGMADIFF outperforms DeepBinDiff in cross-version, cross-optimization-level, and cross-compiler tasks. Besides, we conduct case studies on a patch detection task using real-world CVEs and a vulnerability detection task using Zoom, a popular video conferencing application. Experiments show that SIGMADIFF is able to pinpoint more patches and vulnerabilities than the baselines.

Contributions. This paper makes the following contributions:

- We propose a novel graph neural network-based approach to directly generate token matching results for the pseudocode diffing problem.
- We propose a flow-sensitive, call-site sensitive [69], [70] analysis called *lightweight symbolic analysis* to extract semantic information of a program's high level IRs.
- We improve the design and training of the graph neural network model to effectively solve pseudocode diffing as a graph matching problem.
- We implement a prototype called SIGMADIFF. Our evaluation demonstrates that SIGMADIFF outperforms the state-of-the-art diffing tools at the pseudocode token level. SIGMADIFF can still get accurate results when handling code changes caused by different optimization techniques.

We have made the source code of SIGMADIFF publicly available [2].

II. MOTIVATION

In this section, we first discuss some unique challenges in pseudocode diffing via a motivating example, then go over the existing diffing techniques and their limitations, and finally explain how we tackle these challenges in SIGMADIFF.

A. A Motivating Example

We use a real-world out-of-bounds read vulnerability CVE-2020-13790 [2] from the `start_input_ppm` function in libjpeg-turbo [15] as a motivating example. Figure 1 presents the source code differences between v2.0.4 and v2.1.2, as well as pseudocode diffing results by various diffing tools. The v2.0.4 binary is obtained from Zoom [21], a popular video conferencing COTS (Commercial off-the-shelf) program, and the new version is compiled with GCC v7.5.0 (-O0).

The goal is to precisely identify the token-level changes in pseudocode that are semantically equivalent to the source code-level changes while ignoring the syntactical changes produced in the processes of compilation and decompilation. Here, we list several common yet challenging pseudocode-level changes that are caused by compilers and decompilers:

(1) **Variable renaming.** A variable may be named differently from one version to another. For example, variable `maxval` is named `uVar18` in the old version in Figure 1 (b) but `uVar13` in the new version in Figure 1 (c), shown as the blue boxes.
(2) **Expression merging and splitting.** A decompiler may merge multiple expressions into one, or vice versa. For instance, "`lVar16 + lVar17`" at Line 4 and "`lVar16 = *(long *)(param_2 + 0x48)` at Line 7 in Figure 1 (b) are merged into "`*(long *)(param_2 + 0x48) + lVar14)`" at Line 6 in Figure 1 (c).
(3) **Control structure changes.** A decompiler may represent a loop structure as a `for` loop, a `while` loop, or a `do-while` loop. An `if-then-else` structure may also change, depending on which branch is for "then" or "else". `goto` statements are also highly prevalent in pseudocode. As shown in the purple boxes in Figure 1 (b) and (c), a `while` loop with an `if` statement inside the loop body in the old version is changed to a `do-while` loop in the new version.

B. Existing Techniques

Unfortunately, existing diffing techniques fall short of tackling these pseudocode diffing problems.

String Diffing. Diaphora [9] takes this approach by simply treating two code snippets as two strings and leveraging

[1] SIGMADIFF stands for **S**emant**i**cs-Aware Deep **G**raph **Ma**tching for Pseudocode **Diff**ing.

[2] https://github.com/yijiufly/SigmaDiff

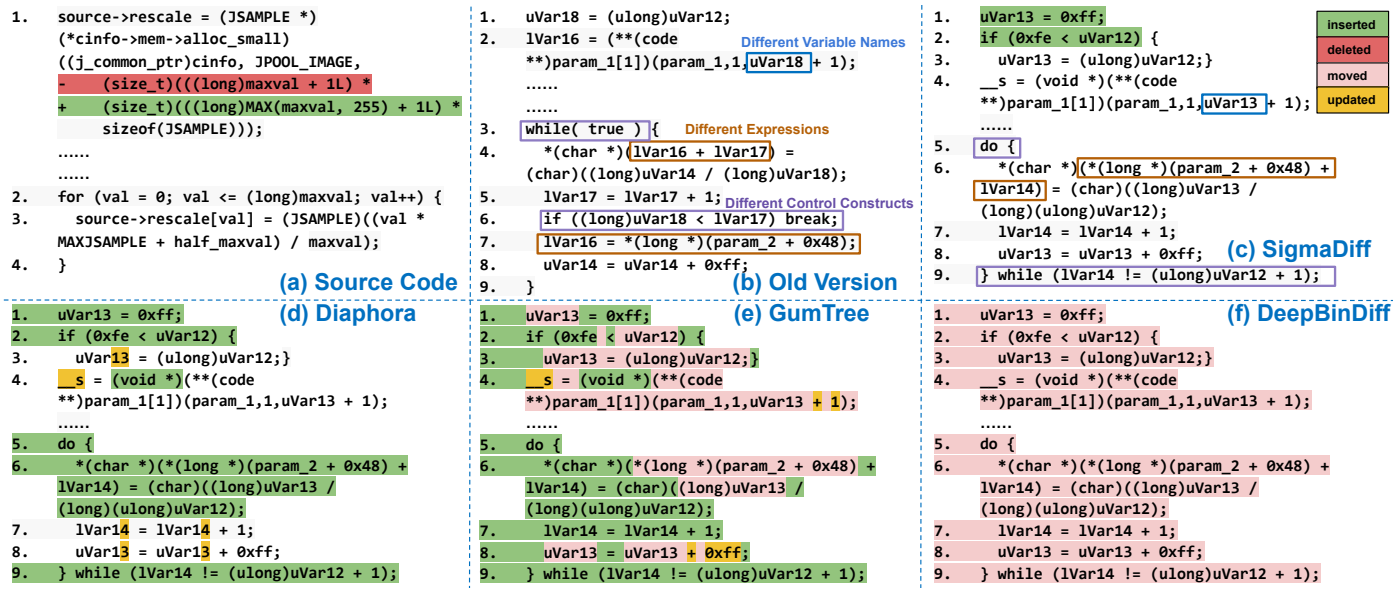


Fig. 1: Motivating example

standard string alignment algorithms to diff the two code snippets. As shown in Figure 1 (d), in addition to identifying the true insertions (Lines 1 and 2), Diaphora also incorrectly recognizes many renamed variables as updates, and expression changes and control structure changes as insertions.

Source Code Diffing. Techniques such as GumTree [41] and ClDiff [52] build abstract syntax trees (ASTs) from the two source code snippets and then match two ASTs. Unlike string diffing, AST diffing ensures that the diffing results always follow the source code grammar. Here, we present GumTree [41] (with the default GreedySubtree and GreedyBottomUp matcher [13]), a widely-used source code diffing tool, in Figure 1 (e). It incorrectly identifies almost all these statements as insertions, and some sub-expressions as moves. While this approach works generally well for source code diffing, it works poorly for pseudocode diffing, because of the substantial amount of syntax-level changes for pseudocode between two versions. In our evaluation in Appendix §E, we also show that the state-of-the-art source code clone detector, NIL [65], is 54% worse than SIGMADIFF on average and is not suitable for pseudocode diffing.

Binary Diffing. Instead of diffing at the pseudocode level directly, we can first perform diffing at the binary level (or more precisely, at the disassembly level), and then map the matching results to the pseudocode level for better readability. We show the diffing results in Figure 1 (f), from a state-of-the-art technique DeepBinDiff's [38]. It mismatches the entire code from Line 1 to 9 with some other code blocks (not displayed in this figure), due to the drastic block-level changes brought by different compilers and compiler optimizations.

C. Our Technique

In the following paragraphs, we discuss how SIGMADIFF handles the aforementioned challenges.

First, we propose to perform diffing at the IR level and then map the results back to the pseudocode level, since we observe that small differences in IRs might lead to significant syntax-level changes in pseudocode. We further perform a lightweight symbolic analysis to associate each IR variable with a symbolic expression that reveals how the value of that IR variable is calculated, so that we can match IR variables by their symbolic expressions instead of their unreliable names. For instance, Line 4 in Figure 1 (b) and Line 6 in Figure 1 (c) are different in pseudocode but similar in IRs. With the help of the generated symbolic expressions (e.g., `lVar16` in the brown box becomes `*(ARG2+0x48)`), SIGMADIFF can easily find the matching.

Second, instead of using the unstable control flow, we rely on the more stable and expressive data and control dependencies to capture contextual information for each IR. Essentially, IRs with similar contexts are likely to be matched. As shown in Figure 1 (b), Line 3-8 are all control dependent on Line 6, and in (c), Line 6-9 are all control dependent on Line 9. Line 6 in (b) and Line 9 in (c) can be matched successfully by looking at the dependencies rather than the unstable and quite different CFGs.

Third, we leverage recent advances in deep graph matching to efficiently solve pseudocode diffing as a graph matching problem. IPDGs can be really huge for real-world programs, and the graph matching problem is NP-Hard [46]. To overcome this challenge, SIGMADIFF leverages the computing power of modern GPUs, and uses the state-of-the-art DGMC model [46]. We design a new loss function that incorporates data and opcode type constraints and propose a "pre-training and fine-tuning" schema to accelerate graph matching.

III. OVERVIEW AND BACKGROUND

A. Approach Overview

The system overview of SIGMADIFF is shown in Figure 2. It consists of three stages: 1) pre-processing, which performs static analyses to represent the binaries as graphs with semantic features; 2) pseudocode diffing, which leverages the DGMC

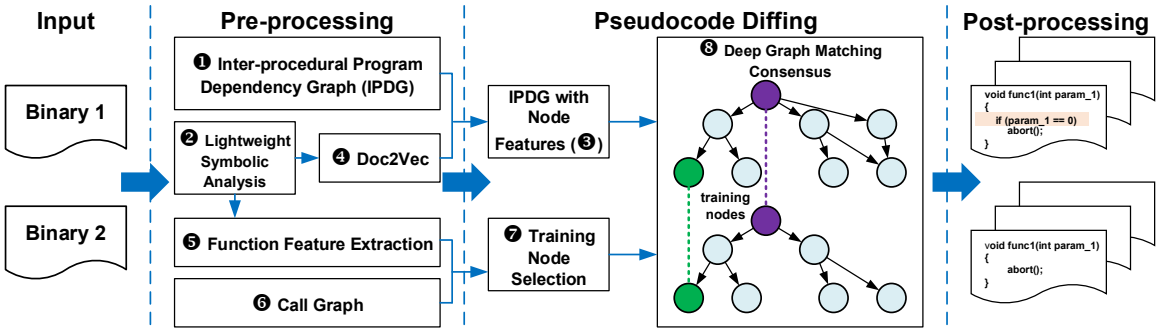


Fig. 2: Overview of SIGMADIFF

model [46] to perform the IR level matching; and 3) post-processing to lift the IR-level matching results to the pseudocode level.

In the *pre-processing* stage, we construct an IPDG (❶ in Figure 2) [45], [51] by analyzing control and data dependencies of the given binary at IR level. We also propose an inter-procedural lightweight symbolic analysis (❷) to extract a symbolic expression for each IR variable. The result is used to: 1) generate node features (❸) (initial embeddings) of the graph model by leveraging the doc2vec (❹) [57] technique; and 2) generate function features (❺) for function-level matching, whose purpose is to help locate training nodes for DGMC (see below). We also construct a call graph (❻) of the binary in the pre-processing stage for function matching. In the *pseudocode diffing* stage, we leverage the DGMC model to obtain the IR-level matching results. In order to locate the training nodes for DGMC's semi-supervised learning, we perform the function-level matching first, so as to narrow down the search space and find out nodes that are unique and representative (❼) within the matched function pairs. Then we run the DGMC model (❽) to obtain the IR-level matching results. We modify the model to make it more suitable for our task. The last stage is *post-processing*, in which we transform the IR-level matching results to pseudocode token diffing results.

B. Background

Symbolic Analysis. The lightweight symbolic analysis we propose is a static analysis that approximates program semantics. It is different from value-set analysis [25] (VSA), an abstract interpretation approach that tracks (an over-approximation of) the set of numeric values each data object holds at each program point. VSA defines abstract locations (*a-loc*s) to represent the data object (whether local, global, or in the heap). The set of numeric values is recorded in strided intervals. Our analysis, instead, uses symbolic expressions to record the value set. Symbolic expressions are more suitable for our task since it records how a value is calculated and thus capture more semantic information than numeric values.

Deep Graph Matching Consensus. DGMC [46] is an end-to-end deep graph matching architecture which consists of two stages. In the first stage, it leverages a Graph Neural Network to generate node embeddings and then obtains the initial matchings according to the embedding similarities. This stage is called "local feature matching", which is a common practice and is similar to other graph matching models (e.g., [24], [81]). However, the embeddings only considered local features, and the model could confuse the correct match with the locally similar ones. Thus, in the second stage, DGMC iteratively eliminates the incorrect matches by ensuring the neighbors of the matched nodes are correctly matched to each other as well, namely achieving *neighborhood consensus*.

IV. PRE-PROCESSING

In the pre-processing stage, we leverage static program analysis to extract the semantics of the two given binaries and construct inputs for the next stage. Specifically, we model the binaries as graphs and extract features of the nodes in the graphs as well as the functions in the binaries. The graphs and features will be fed into the pseudocode diffing stage.

A. Graph Construction

We generate an IR-level IPDG [45], [51] to represent the whole binary in order to capture the contextual information for each IR. Specifically, a node in the graph represents an IR statement, and an edge represents a control or data dependency relationship. In the following, we will frequently refer to the IR node as IR for brevity.

Figure 4 shows a pseudocode snippet and its corresponding IPDG. Each white node in Figure 4 represents a Ghidra IR, formatted as "`line# output opcode input0 [, input1 ...]`". It is worth noting that while our current implementation builds atop Ghidra IR, our design is generic enough to support any high-level IRs. For instance, we have confirmed that the steps in our workflow can also be performed on LLVM IRs [16] lifted by RetDec [19].

We follow the standard way [51] to handle the callsites in IPDG. Specifically, we add entry, argument, and return nodes for both two functions, and `CALLSITE_RET` and `CALLSITE_ARG_n` ($n = 1, 2, \ldots$) nodes at callsites (node 3 in the example). They are marked as gray nodes in Figure 4. These nodes do not represent any IR, but summarize the data and control dependencies between functions.

Note that to reduce the complexity of graph matching (§V-A), we choose to treat control and data-dependency equally and treat these dependencies as directed edges.

B. Semantic Features Extraction

We conduct a lightweight inter-procedural symbolic analysis (similar to [1], [4]) to extract node features, which are then used as the initial node embeddings and to construct function features.

1) Lightweight Symbolic Analysis: The outputs of lightweight symbolic analysis are the symbolic expressions that each IR variable holds at each program point. In this section, we will first explain the IR syntax. Then we formalize the lightweight symbolic analysis by showing the definitions of symbolic expression and the interpretation function that interpret IR variables into symbolic expressions. Next, we describe how we conduct inter-procedural analysis. Lastly, an example is presented.

IR Syntax. An abbreviated definition of the Ghidra high-level IR syntax is shown in Table I. Most statements are self-explanatory. In particular, `STORE` stores $expr_2$ to the destination address pointed by $expr_1$. `CBRANCH` takes the first expression as a condition and jumps to the location pointed by $expr_2$. `CALL` calls the function at $expr_1$ (with zero or more parameters and zero or one return value). `LOAD` is to load from the memory specified by $expr$. `MULTIEQUAL` is the phi-node in SSA form [31]. It merges expressions $expr_1, expr_2, ...$, from different paths.

TABLE I: An abbreviated Ghidra IR Syntax

$program$	::= $stmt*$
$stmt$	::= `STORE` $expr_1, expr_2$ | `BRANCH` $expr$ | `RETURN` $expr$ | var := $expr$ | `CBRANCH` $expr_1, expr_2$ | $expr$ `CALL` $expr_1, ...$
$expr$	::= var | $\diamond_U expr$ | $\diamond_B$ $expr_1, expr_2$ | $\diamond_{cmp}$ $expr_1, expr_2$ | `LOAD` $expr$ | `MULTIEQUAL` $expr_1, expr_2, ...$
$\diamond_U$	::= `INT_NEGATE` | `BOOL_NEGATE` | ...
$\diamond_B$	::= `INT_ADD` | `INT_SUB` | `INT_MULT` | `INT_DIV` | ...
$\diamond_{cmp}$	::= `INT_NOTEQUAL` | `INT_EQUAL` | ...

Symbolic Expression. A symbolic expression approximates the value of a high-level IR variable. We define the syntax of symbolic expressions se in Figure 3 and the operations on se in Table II.

$$
\begin{aligned}
se &::= se_{formula} \mid se_{source} \\
se_{formula} &::= const \mid source \mid [se_{formula}] \mid \diamond_u se_{formula} \mid \\
&\quad\ se_{formula1} \diamond_b se_{formula2} \\
se_{source} &::= \emptyset \mid \{source\} \mid \{source_1, source_2, ...\} \\
source &::= Str \mid Symbol \mid ARGn \\
\diamond_u &::= \neg \mid \sim \\
\diamond_b &::= + \mid - \mid * \mid \div \mid and \mid or \\
const &::= ..., -1, 0, 1, 2, ...
\end{aligned}
$$

Fig. 3: Definitions of Symbolic Expression

Essentially, the symbolic expression is either a simple formula $se_{formula}$ or a set of sources se_{source}. Note that $se_{formula}$ is in a nested structure, where existing expressions can form a new one starting from basic expressions such as $ARGn$ ($n \in \mathbb{N}_+$) and $const$ ($const \in \mathbb{Z}$). In particular, $[se_{formula}]$ represents the memory location pointed by $se_{formula}$. se_{source} is defined to represent the merge result of symbolic expressions. Merging happens when a variable could be equal to multiple symbolic expressions (depending on which control-flow is executed). Ideally, we would like to keep all expressions when merging, but it is impossible since the size of expressions will grow exponentially. Thus, we pick the most representative origins of the expressions (i.e., $source$) to be semantic-aware while still being computationally feasible. This `merge` operation is described in Table II. Compared to only keeping one expression when merging [1] (an existing open-source implementation), our approach is able to cover all the control-flows so that the results are not biased towards one branch or another. We use this operation when two branches are merged or a function has multiple returns. It is also helpful for handling loop variants: we will traverse the loop iteratively until a fixed point is reached (i.e., no further changes are made to the expressions). Note that $\texttt{sources}(se)$ and se can be partially ordered such that $se \subseteq \texttt{sources}(se)$. The `binop` and `merge` operations in Table II and the interpretation rules in Table III are monotonic. Thus, a fixed point is guaranteed to be reached [73].

TABLE II: Descriptions of Symbolic Operations

Operations	Descriptions
$\texttt{binop}(se_1, se_2, \diamond_b)$	It equals to $se_1 \diamond_b se_2$ if $se_1, se_2 \in se_{formula}$. Otherwise, it equals to $\texttt{merge}(se_1, se_2)$. Noted that we will calculate the numeric value of $se_1 \diamond_b se_2$ if $se_1 se_2 \in const$.
$\texttt{sources}(se)$	The source set of se. $\texttt{sources}(se)$ ::= $\{s_1, ..., s_n\}$, where s_i is the smallest symbolic expressions in se that cannot be split up, and $s_i \in \{ARGn, Str, Symbol\}$. E.g., if $se = ARG1 + ARG2 + 8$, $\texttt{sources}(se) = \{ARG1, ARG2\}$.
$\texttt{merge}(se_1, se_2, ...)$	The merged source set of $se_1, se_2, ...$ It equals to $\texttt{sources}(se_1) \cup \texttt{sources}(se_2) \cup ...$, which is the set union result.

Interpretation Function. We define a function α that interprets an IR variable/expression as a symbolic expression.

In general, symbolic expressions are calculated recursively. The symbolic expressions of string, global symbol, constant values, and arguments can be determined directly. Strings, global symbols, and arguments are directly interpreted as the symbols Str, $Symbol$, and $ARGn$ ($n \in \mathbb{N}_+$). Constant values are interpreted as the number $const$. In other cases, we choose the corresponding interpretation rule shown in Table III to calculate the symbolic expressions for different IRs.

The first five rules in Table III are designed for interpreting expressions. For unary or binary expressions, the interpretation is straightforward. $\diamond_{cmp}$ expression is interpreted as 0 on the false branch or 1 on the true branch of the following $CBRANCH$ statement. For `LOAD` expression, we first interpret $expr$ to get the address and load values from the address. For `MULTIEQUAL` expression, we leverage the `merge` operation to merge the source set of different symbolic expressions. The last two rules are for interpreting IR statements. The symbolic expression of var is equal to $expr$ in the assignment statement. In `store` statement, we store the symbolic expression of $expr_2$ to the memory location pointed by $\alpha(expr_1)$. Note that these rules are calculated recursively. In the end, the $expr$ cannot be further split up and we directly interpret them as either $const$, Str, $Symbol$, or $ARGn$ ($n \in \mathbb{N}_+$).

Inter-procedural Analysis. We devise an algorithm to conduct the inter-procedural analysis, as shown in Algorithm 1. For better efficiency, functions in the call graph are visited in post-order (Line 2), so that we only go over each function once. After analyzing each function (Line 6), we will merge

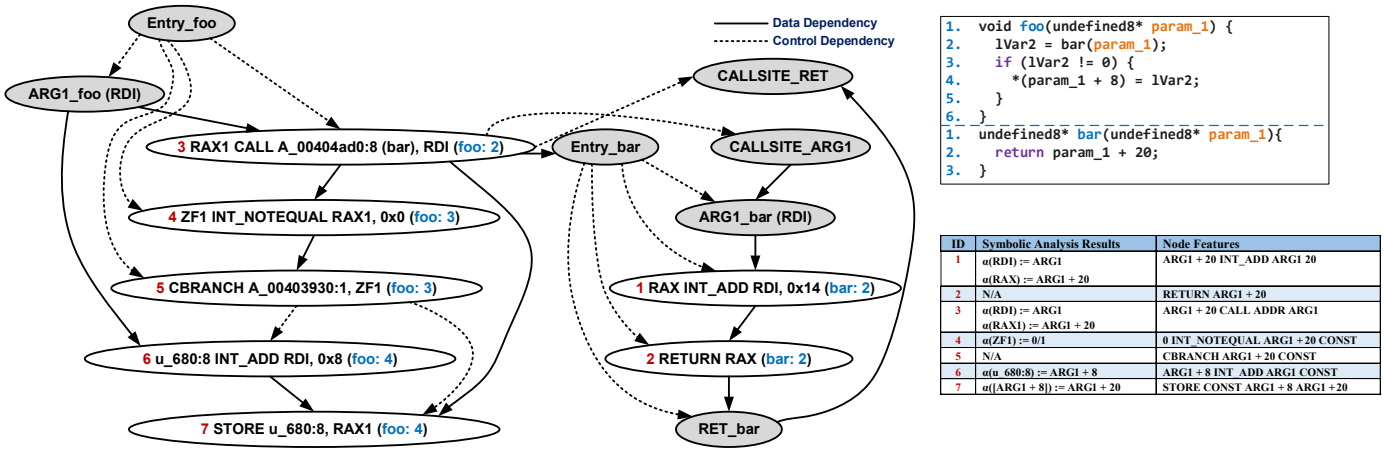


Fig. 4: An example of program dependency graph, lightweight symbolic analysis, and node feature extraction

TABLE III: Interpretation Rules

IR Expression/Statement	Actions
$\diamond_U expr$	$\alpha(\diamond_U expr) \coloneqq \diamond_u \alpha(expr)$
$\diamond_B\ expr_1, expr_2$	$\alpha(\diamond_B\ expr_1, expr_2) \coloneqq \texttt{binop}(\alpha(expr_1), \alpha(expr_2), \diamond_b)$
$expr_1\ \diamond_{cmp}\ expr_2$	$\alpha(expr_1\ \diamond_{cmp}\ expr_2) \coloneqq 0/1$
LOAD $expr$	$\alpha(\texttt{LOAD}\ expr) \coloneqq [\alpha(expr)]$
MULTIEQUAL $expr_1, expr_2, ...$	$\alpha(\texttt{MULTIEQUAL}\ expr_1, expr_2, ...) \coloneqq \texttt{merge}(\alpha(expr_1), \alpha(expr_2), ...)$
$var \coloneqq expr$	$\alpha(var) \coloneqq \alpha(expr)$
STORE $expr_1, expr_2$	$[\alpha(expr_1)] \coloneqq \alpha(expr_2)$

its return values if there are multiple returns (Lines 7 and 8). The return values of every analyzed function are recorded (Line 9) in a dictionary $gRetVal$. When analyzing a callsite, to integrate the calling context, we first retrieve the return values of the callee function from $gRetVal$, and then update the return value expressions according to the arguments that are passed to the callee function. For instance, if callee's return value is ARG1+100 (ARG1 represents the first argument of the callee), and the first argument passed to it is ARG2−10 (ARG2 represents the second arguments of the caller), we update the return value to be ARG2+90. That being said, our inter-procedural analysis is call-site sensitive [69], [70], as return values of the same function from different call sites vary. Lines 10 to 11 in Algorithm 1 will be introduced later.

Algorithm 1 Inter-procedural Analysis

procedure LightweightSymbolicAnalysis($binary$)
1: $cg \leftarrow$GenCallGraph($binary$)
2: $workQueue \leftarrow$postOrderDFS(cg)
3: $gRetVal \leftarrow \emptyset$
4: **while** $workQueue! = empty$ **do**
5: $func \leftarrow workQueue$.pop()
6: $retVals, symVals \leftarrow$ VisitFunc($func, gRetVal$)
7: **if** $len(retVals) > 1$ **then**
8: $retVals \leftarrow$merge($retVals$)
9: $gRetVal[func] \leftarrow retVals$
10: ExportNodeAndFuncFeatures($func, symVals$)
11: ExportCG(cg)
end procedure

Example. Using the pseudocode snippet in Figure 4, we show the symbolic analysis results in the second column of the bottom right table. We start at the callee function bar. First, we initialize the expression for parameters. RDI is denoted as ARG1 since it is in an x86-64 binary. The first row in the table shows the symbolic analysis results after the INT_ADD IR ($ID = 1$). It is a binary operation and we calculate the symbolic expression of RAX to be ARG1+20. The second row is the RETURN IR ($ID = 2$) in function bar. Thus we record the returned value ARG1+20 and clear the context. Then we start to analyze the first IR ($ID = 3$) in foo function, in which the function bar is called with a parameter RDI. We refer to the stored return value of *bar* and compute the return value to be ARG1+20 since the parameter passed to this function happens to be ARG1. The fourth row ($ID = 4$) shows that the output of comparison-related IRs is 0 on the false branch or 1 on the true branch. The seventh row shows an example for STORE ($ID = 7$), in which we record that the value stored at address ARG1+8 at this line of IR is ARG1+20. Another example that contains merge operation is provided in Appendix §B, along with the soundness analysis.

2) Node Feature Extraction: As discussed in §II-C, in order to solve the variable renaming problem and capture the semantics of the IRs, we leverage the output of lightweight symbolic analysis to generate node embeddings that are fed into the graph matching model as the initial node features. Therefore, after lightweight symbolic analysis, we go over all the function IRs once again (Line 10 in Algorithm 1), replace each variable with its symbolic expression, keep the opcode, and use the expression as the node feature for this IR.

The third column in the table in Figure 4 shows the node features we generated. As shown in the fourth row, we normalize the constant 0x0 to CONST, while keeping the constant number in ARG1+20 since it is involved in an expression. Strings and addresses are also normalized. For a conditional jump IR (CBRANCH, $ID = 5$), we backtrack its previous IR and append the input variables of the Status Flag (the Zero Flag ZF1 in this example, its inputs are RAX1 and 0x0, $ID = 4$) to it to reflect the jump condition.

After dealing with all the IRs, we build a corpus using all the IR expressions within the two compared binaries and then leverage doc2vec [57] to learn the embedding of each

expression, which is the initial feature of each IR (i.e., node). The vocabulary of doc2vec corpus is limited since the vocabulary of symbolic expressions (according to their definitions in Figure 3) is small. To improve efficiency, we pre-train the model with the corpus from one binary, then generate initial embeddings for other binaries by using the model directly. Please refer to §V-C for more details on this pre-training idea.

3) Function Feature Extraction and Call Graph Generation: We also leverage the results of lightweight symbolic analysis to extract the function features, which will be helpful for function matching and training node selection (Line 10 in Algorithm 1). We choose the function features for function matching rather than representation learning-based approaches (e.g., Asm2Vec [37] and PalmTree [58]) because we are not conducting a binary function similarity detection [50] in this step. The main purpose of function matching in SIGMADIFF is to locate the training nodes with high quality, therefore, our aim is to find out the features that can represent the unique characteristics of the function and are more stable than syntax facing various code transformations. Specifically, we extracted the following function features: *Return values*, *Side effects*, *Loads*, *Strings and library calls*, and *Function parameter types*. More details about these features are discussed in Appendix §C.

During the analysis, the call graph of the whole binary has also been stored (Line 11 in Algorithm 1) to support the later training nodes selection (in the function-level matching step, see §V-B). We simply ignore indirect calls since we can still select training nodes and perform whole binary matching when the call graph is incomplete. The influence of indirect calls is evaluated in §VI-B5 by testing on C++ programs. Apart from the nodes that represent functions, we also add virtual nodes that represent strings and external functions to the call graph.

V. PSEUDOCODE DIFFING

Given the IPDGs at IR level with initial embeddings and the function feature sets generated in the pre-processing stage, SIGMADIFF then performs pseudocode diffing. Specifically, it first searches for node pairs with high similarity and uniqueness so that we are confident that these pairs should be matched. SIGMADIFF treats these node pairs as training nodes and performs semi-supervised learning that uses the Deep Graph Matching Consensus (DGMC) [46] model to generate IR matching results.

A. Graph Matching Consensus Model

The DGMC model [46] is a two-stage neural architecture that aims to reach a neighborhood consensus [67] when solving the graph matching problem. We introduce the details of the two stages in this section. In the first stage, it generates the initial matching results. To do so, source graph $G_s = (V_s, A_s, X_s, E_s)$ and target graph $G_t = (V_t, A_t, X_t, E_t)$ are fed into a shared graph neural network Ψ_{θ_1}, where V, A, X, and E denote the nodes, adjacency matrix, node features, and edge features (in this paper, we treat edges equally) respectively. Then the latent node embeddings H_t and H_s are generated for both graphs through Ψ_{θ_1}. And the initial soft correspondence $S^{(0)}$ are calculated using the latent node embeddings as follows: $S^{(0)} = sinkhorn(H_s H_t^T) \in [0,1]^{|V_s| \times |V_t|}$, where $sinkhorn$ [71] is a normalization function. The i-th row vector $S^{(0)}_{i,:} \in [0,1]^{|V_t|}$ indicates the probability distribution over i being matched with nodes in G_t for each node $i \in V_s$. During training, DGMC minimizes the loss $\mathcal{L}^{(initial)} = -\sum_{i \in V_s} log(S^{(0)}_{i,\pi_{gt}(i)})$, where π_{gt} denotes the ground truth and the loss function $\mathcal{L}^{(initial)}$ represents the difference between initial matching results and the ground truth.

In the second stage, DGMC leverages another shared graph neural network Ψ_{θ_2} to detect violations of neighborhood consensus in previous matching results. This refinement step is done iteratively so that the neighborhood consensus can be improved through L iterations. In general, the source and target graph with node coloring as node features are fed to Ψ_{θ_2}: $O_s = \Psi_{\theta_2}(I_{|V_s|}, A_s, E_s)$ and $O_t = \Psi_{\theta_2}(S^T_{(l)} I_{|V_s|}, A_t, E_t)$. $S_{(l)}$ is the soft correspondence matrix in the l-th iteration, $l \in 0, ..., L$. The identity matrix $I_{|V_s|}$ and the $S^T_{(l)} I_{|V_s|}$ can be viewed as the node coloring for G_s and G_t, since $S_{(l)}$ is a map from the node function space in the source graph to the node function space in the target graph. Then the vector $d_{i,j} = \vec{o_i^{(s)}} - \vec{o_j^{(t)}}$ that measures the neighborhood consensus between node pairs of these two graphs can be used to update S together with a multi-layer perceptron (MLP) Ψ_{θ_3}: $S^{(l+1)}_{i,j} = sinkhorn(\hat{S}^{(l+1)})_{i,j}$ with $\hat{S}^{(l+1)}_{i,j} = \hat{S}^{(l)}_{i,j} + \Psi_{\theta_3}(\vec{d_{j,i}})$. The loss function in step two is $\mathcal{L}^{(refine)} = -\sum_{i \in V_s} log(S^{(l)}_{i,\pi_{gt}(i)})$.

Evaluations show that the performance of the DGMC model is good even when there is structural noise in graph matching [46], which benefits from applying matching consensus in the second stage.

B. Training Node Selection

SIGMADIFF learns the DGMC model in a semi-supervised fashion. Given a pair of source and target graphs and a small set of training nodes that have already been matched, the model will learn mappings for the rest of the nodes between the two graphs. The intuition is that a small set of nodes (or IRs) in the source graph can perfectly match with a set of nodes in the target graph with very high confidence. These initial matching pairs can help match the nearby nodes in these two graphs by considering the neighborhood consensus and they are the training nodes for this semi-supervised learning.

Therefore, obtaining a fair amount of high-quality training nodes is crucial. We design a two-step strategy for this. First, we conduct a conservative function-level matching to narrow down the search scope. We will find more unique IRs within the scope of a function than within the scope of the whole binary. Note that the only purpose of the function-level matching is for training node selection, the matching of the rest of the nodes is still performed on the whole binary. After the function-level matching, we search for unique IR pairs within the matched function pairs. We introduce the details of these two steps in the remainder of this subsection.

1) Function-level Matching: In this step, we define a measurement that evaluates the similarity between two functions using the function feature sets extracted in the pre-processing stage. Specifically, we calculate a similarity score between two functions according to their feature sets, which include

Return values, *Side effects*, *Loads*, *Strings and library calls*, and *Function parameter types*.

After obtaining the function similarity scores, we leverage a 2-hop greedy matching algorithm to conduct a function-level matching. We use the same *k*-hop algorithm in DeepBinDiff [38]. More details are introduced in Appendix §C.

2) Training Node Selection: As mentioned earlier, in this step we search for IR pairs that have unique features within the scope of the matched functions. The feature we consider is the same feature generated in §IV-B, which is the IR associated with symbolic expressions. We first find out the IRs that have unique features in each function. For instance, all IRs in Figure 4 have unique features. In practice, there will be more duplicate node features, making the occurrence of unique IRs rarer. These IRs form a "Unique IR Set". Then, for each matched function pair, we select the unique IRs they share as the training nodes. In other words, the training nodes are in the intersection set of the two Unique IR Sets.

C. Whole Binary Matching

SIGMADIFF then leverages the DGMC model to perform whole binary matching. We do not perform matching function by function because *function inlining* can change the boundaries of functions. Given the training nodes and two IPDGs, this step creates a near-optimal node mapping that considers both IRs' local features and their neighbors. To better suit DGMC to our problem, we have made the following improvements.

First, we leverage the recovered *data type and opcode type constraints* to further boost the matching accuracy. The intuition is as follows: IRs that operate on different data types cannot be matched together, and neither can the IRs that have different types of opcode. To encode these constraints into the model, we add two terms to the original loss function:

$$\mathcal{L} = \mathcal{L}^{(initial)} + \mathcal{L}^{(refine)} + \alpha\mathcal{L}^{(data\_type)} + \beta\mathcal{L}^{(op\_type)} \quad (1)$$

with

$$\mathcal{L}^{(data\_type)} = \sum_{j \in W_s} (e^{S^{(l)}_{j,\pi(j)}} - 1) \quad (2)$$

$$\mathcal{L}^{(op\_type)} = \sum_{k \in W'_s} (e^{S^{(l)}_{k,\pi(k)}} - 1) \quad (3)$$

$W_s \subseteq V_s$ in equation (2) denotes the nodes in the source graph that have an incompatible data type matching, similarly, $W'_s \subseteq V_s$ in equation (3) denotes the nodes that have an incorrect matching in terms of the opcode types. In Equation (2) , $\pi(j)$ denotes the top-1 candidate node that j is matched to $\forall j \in V_s$. α and β in Equation (1) are the coefficients for $\mathcal{L}^{(type)}$ and $\mathcal{L}^{(value)}$ respectively. We use the exponential function to increase the penalty of incompatible types in the loss function.

Type compatibility is determined by a type lattice. We show an abbreviated type lattice of the x86_64 programs in Ghidra in Figure 5. Here, $\top$ includes `undefined`, `undefined *`, and `void *`. However, since decompiler often cannot infer the types correctly, we cannot strictly follow the subtype constraints exerted by the type lattice. We relax the constraints

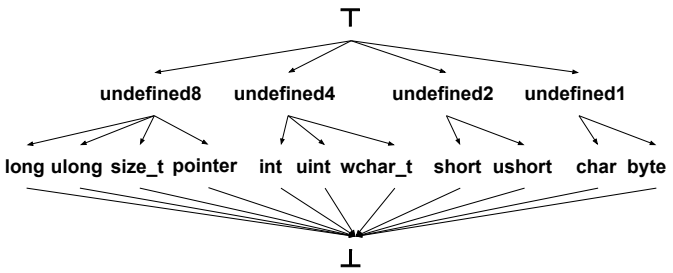


Fig. 5: An abbreviated type lattice of x86_64 in Ghidra

in the following two cases. First, decompilers often cannot correctly infer signed and unsigned types, thus we make signed and unsigned types compatible with each other (e.g., `int` vs. `uint`). Similarly, `char` and `byte` are also compatible. Additionally, all the pointer types are considered compatible with each other. For opcode types, comparison-related IR operations are considered compatible in order to tolerate control-flow changes.

Second, we add *more hops* when building the first neural network Ψ_{θ_1} (see §V-A), since our graphs are usually very sparse and we want to cover a larger scope for each IR so that more neighborhood contexts are considered.

Third, we introduce the *"pre-training and fine-tuning"* schema in order to further improve the efficiency of SIGMADIFF. Specifically, we pre-train the DGMC model with a pair of large binaries with a large epoch value (800 epochs in this paper). The doc2vec model is also pre-trained on this pair of binaries. Then for the program pair under analysis, we further fine-tune the pre-trained DGMC model with only a few more epochs (200 epochs in this paper). We use the early-stopping technique in both steps in order to ensure efficiency and avoid over-fitting. The pre-training process can be executed in advance and offline. In the inference stage of SIGMADIFF, we only need to execute the fine-tuning process if a pre-trained model is available. This schema greatly improves the efficiency of SIGMADIFF.

Fourth, we design an *iterative algorithm* to avoid the out-of-memory problem in GPU when matching large binaries. In each iteration, we feed one pair of matched functions (obtained from function-level matching results §V-B1) to generate diffing results for them. We then reduce these two functions into two nodes in their IPDGs. We repeat this process until the reduced IPDGs can fit in the GPU.

D. Post-processing

SIGMADIFF then prunes the IR matching results using the *type constraints* (see §V-C) and selects the top-1 from the pruned candidates by referring to the matrix S. Since each IR is mapped to a group of tokens in pseudocode, SIGMADIFF is able to generate the pseudocode matching results at the token level using the IR matching results. In other open-source decompilers, since high-level IR is an intermediate result in decompilation, the mapping from high-level IR to pseudocode can also be obtained through some engineering work.

To further match the two groups of tokens associated with the matched IRs, we could utilize the IR variables and match

TABLE IV: IRs and associated tokens in the motivating example

IRs	Associated tokens in Figure 1 (b)	Associated tokens in Figure 1 (c)
RAX INT_ADD RSI, 0x48	RSI → param_2 (at Line 7) 0x48 → 0x48 (at Line 7) RAX → param_2 + 0x48 (at Line 7)	RSI → param_2 (at Line 6) 0x48 → 0x48 (at Line 6) RAX → param_2 + 0x48 (at Line 6)
u_100 INT_ADD u_120, u_200	u_120 → lVar16 (at Line 4) u_200 → lVar17 (at Line 4)	u_120 → *(long *)(param_2 + 0x48) (at Line 6) u_200 → lVar14 (at Line 6)

their corresponding tokens. In our motivating example, the IRs for Line 4 and 7 in Figure 1 (b) and for Line 6 in Figure 1 (c) are similar, and SIGMADIFF can match them correctly. We include two IRs in Table IV, in which the IR variables (and corresponding tokens) we could match are `RSI`, `0x48`, `u_120`, and `u_200`. The mapping from IR variables to tokens can either be obtained directly from Ghidra (e.g., `RSI`, `0x48`, and `u_200`), or be derived based on IR variables' definition (e.g., `RAX`).

Some tokens (e.g., indents, brackets, parentheses, and commas) are not mapped to any IR. They exist in pseudocode only for formatting purposes and do not carry semantic meanings. SIGMADIFF does not generate results for them.

VI. EVALUATION

In this section, we begin by describing the experimental setup and evaluating the effectiveness of SIGMADIFF in cross-version, cross-optimization-level, cross-compiler, cross-architecture, and C++ diffing tasks. Next, we evaluate the efficiency of SIGMADIFF. Furthermore, we perform an ablation study. We also show the security applications of SIGMADIFF by conducting two large-scale case studies.

A. Experimental Setup

The training and testing of SIGMADIFF (and the baseline models' evaluations) are conducted on a dedicated server with a Ryzen 3900X CPU@3.80 GHz×12, one RTX 2080Ti GPU, 64 GB memory, and 500 GB SSD. We implemented the lightweight symbolic analysis and IPDG generation functionalities based on Ghidra v9.2.2's APIs. We developed the lightweight symbolic analysis technique after referencing two open-source implementations [1], [4].

1) Deep Learning Model Settings: The implementation of the DGMC model is based on the package[3] released by its authors [46]. We choose the following hyperparameters (detailed evaluations are reported in Appendix §A): the dimension of the initial node embeddings generated by doc2vec is 128; dimension of the hidden states of Ψ_{θ_1} is 128 (Ψ_{θ_2} is 32); the number of layers of Ψ_{θ_1} and Ψ_{θ_2} is 3; the number of hops of Ψ_{θ_1} is 3 (Ψ_{θ_2} is 1); training epochs for pre-training is 800; training epochs for fine-tuning is 200; early-stopping patience is 30 epochs; the optimizer is Adam [54] with a learning rate of 0.001; $\alpha = \beta = 0.1$.

2) Datasets: We compiled different versions of the source code of five popular binary/binary sets: Coreutils [3], Diffutils [10], Findutils [11], GMP [12], and Putty [18]. Specifically, we compiled three versions of Coreutils (v5.93, v6.4, v8.1), three versions (v2.8, v3.4, v3.6) of Diffutils, three versions (v4.2.33, v4.4.1, v4.6.0) of Findutils, three versions (v6.0.0, v6.1.1, v6.2.1) of GMP, and three versions (v0.75, v0.76, v0.77) of Putty using GCC v5.4 (same as DeepBinDiff [38]) with four optimization levels (O0, O1, O2, O3). In total, there are 1,224 binaries. We utilize the debug symbols to collect the ground truth, but all our experiments are conducted on the stripped binaries. Our model is pre-trained using the `not` binary in the LLVM (v3.7.0 and v3.8.1) framework's implementation. We have verified that this binary has no overlapping functions with our dataset. Note that we are showing a lower bound by pre-training on a completely different binary. This strategy can test SIGMADIFF's generalizability.

Ground Truth. Even though we conduct diffing at the pseudocode token level, we do not have the precise ground truth at such a granularity level. Instead, we perform workarounds to estimate the effectiveness. The pseudocode tokens are mapped to program addresses through Ghidra. The program addresses are mapped to the source code file names and line numbers by running `addr2line` [5] (debug symbols are required). Then, given two matched pseudocode tokens, the source code file names and line numbers for them are retrieved and checked to see if their lines are matched.

It is worth noting that the preceding process is only used to get the ground truth. In reality, we will not have the source code files for the binaries under analysis.

For the source code of different versions, we extract the line-level matching ground truth in the same way as DeepBinDiff [38]. Specifically, we use the Myers algorithm [63] to perform text based matching and only consider the lines that are identical to ensure the correctness of the ground truth.

3) Baseline Techniques: We consider Diaphora (>2k stars on GitHub) [9] and DeepBinDiff (state-of-the-art deep learning-based binary diffing tool) [38] as the baselines for comparison. Diaphora only supports IDA Pro [14], while our implementation relies on Ghidra. For a fair comparison, we first run Diaphora on IDA Pro (v7.5) to obtain the function-level matching results, then get the pseudocode of these functions from Ghidra. Note that we only consider the intersection of the functions identified by Ghidra and IDA Pro in order to achieve a fair comparison since they are not exactly the same. Then we run the pseudocode diffing algorithm of Diaphora to diff the pseudocode of the matched function pairs. Diaphora produces four result types: matched, changed, added, and deleted. We consider matched and changed token pairs as the final matched token pairs. And we only consider tokens that are contained in the ground truth even though Diaphora outputs the diffing results for all the tokens.

DeepBinDiff [38] is designed for basic-block level diffing. In order to (directly) compare with it, we convert its diffing results to pseudocode token level. Specifically, we consider all the tokens in a basic block are correctly or incorrectly matched if the basic block is correctly or incorrectly matched.

[3]https://github.com/rusty1s/deep-graph-matching-consensus

By doing so, the total token set of Diaphora, DeepBinDiff, and SIGMADIFF will be the same, which is all the pseudocode tokens that are contained in the ground truth.

4) Evaluation Metrics: We use *precision*, *recall* and *F1-score* to evaluate the diffing results on the source graph. Moreover, we consider the following principles when performing evaluation: 1) we only consider the token in the source binary that has a valid match in the ground truth; and 2) if a token in the source binary is matched to multiple token targets according to the ground truth, we consider the token is matched correctly as long as one of the targets is found correctly since we do not aim to solve the many-to-many matching in this paper (we further discuss this in §VIII).

B. Effectiveness

We conduct four sets of experiments to evaluate the effectiveness of SIGMADIFF. Specifically, we compare SIGMADIFF, Diaphora, and DeepBinDiff on cross-optimization-level, cross-version, and cross-compiler binaries; SIGMADIFF and Diaphora on cross-architecture binaries since DeepBinDiff only supports x86 binaries.

1) Cross-version Diffing:

In this experiment, we compare the performance of SIGMADIFF, DeepBinDiff, and Diaphora in cross-version diffing tasks. We test them on different versions of binaries in Coreutils, Diffutils, Findutils, Gmp, and Putty, by comparing the lower versions of the binaries with the higher versions. All of the binaries are compiled with their default optimization level. The results are shown in Table V.

In general, SIGMADIFF outperforms DeepBinDiff and Diaphora in all of the tasks. Since all tokens in a block are treated as correctly matched as long as the block is correctly matched, DeepBinDiff will benefit from this evaluation strategy. Nevertheless, SIGMADIFF outperforms DeepBinDiff by 5.6% on Coreutils, 2.9% on Diffutils, and 10.5% on Findutils in terms of F1-score. Gmp and Putty are too large for DeepBinDiff to process (for instance, it needs days to analyze Putty, thus the "N/As" in the table). We found that when there are strings or library calls nearby, DeepBinDiff can match the blocks with high accuracy, but when there are no strings or library calls, the accuracy is worse. In contrast, SIGMADIFF relies on unique IRs, which include strings and library calls, and other useful training nodes (an investigation on the number of training nodes is shown in Appendix §D). Additionally, since DeepBinDiff uses *k*-hop block matching, it could mismatch surrounding blocks that are similar.

Moreover, SIGMADIFF outperforms Diaphora by a large margin when the differences between versions are considerable. For instance, SIGMADIFF outperforms Diaphora by 43% on Diffutils, 43% on Findutils, and 25% on Coreutils on average in terms of F1-score. In general, the string-based diffing algorithm in Diaphora will match conservatively and label a difference that includes multiple tokens or statements as added or removed. Therefore, Diaphora's precision is much higher than its recall in all of the ten tasks. Additionally, Diaphora can produce precise diffing results when the differences between versions are small (such as Gmp and Putty).

2) Cross-optimization-level Diffing:

We then conduct experiments to test the performance of SIGMADIFF, DeepBinDiff, and Diaphora in cross-optimization-level diffing tasks. The O0 and O3, O1 and O2, O2 and O3, O2 and O3 binaries in Coreutils, Diffutils, Findutils, Gmp, and Putty are compared. Figure 6 presents the Cumulative Distribution Function (CDF) figures of the F1-scores. Note that we merge the F1-scores for diffing the Coreutils, Diffutils, and Findutils binaries' different versions and draw the three figures. Gmp and Putty are not included in these figures since these binaries took too long for DeepBinDiff to process. But we show all the detailed results of each task (including Gmp and Putty) in the artifact repository.

As shown in Figure 6, for cross-optimization-level diffing, SIGMADIFF significantly outperforms Diaphora and DeepBinDiff, especially when the optimization level gap becomes larger. We observed that Diaphora fails to match a large number of tokens when the control constructs are different (e.g., if and else branches are switched). On average, SIGMADIFF outperforms Diaphora by 308%, 85%, 38% in terms of F1-score for O0 vs. O3, O1 vs. O3, and O2 vs. O3 respectively. DeepBinDiff can match against optimizations such as function inlining, but it will miss the tokens when the optimizations merge two basic blocks or significantly change the control flow graph structure. As shown in Figure 6a, DeepBinDiff gets similar results as Diaphora in O0 vs. O3 diffing tasks. These results show that SIGMADIFF is more resilient to optimization level changes compared to Diaphora and DeepBinDiff, meaning its matching strategy is indeed beneficial.

3) Cross-compiler Diffing:

We also conduct experiments between GCC and Clang binaries on SIGMADIFF, Diaphora, and DeepBinDiff to show SIGMADIFF's ability in cross-compiler diffing. More specifically, we compile all the binaries in Coreutils 8.1, Diffutils 3.6, Findutils 4.6, Gmp 6.2.1, and Putty 0.76 with GCC (v5.4.0) and Clang (v3.8.0) using the default optimization settings.

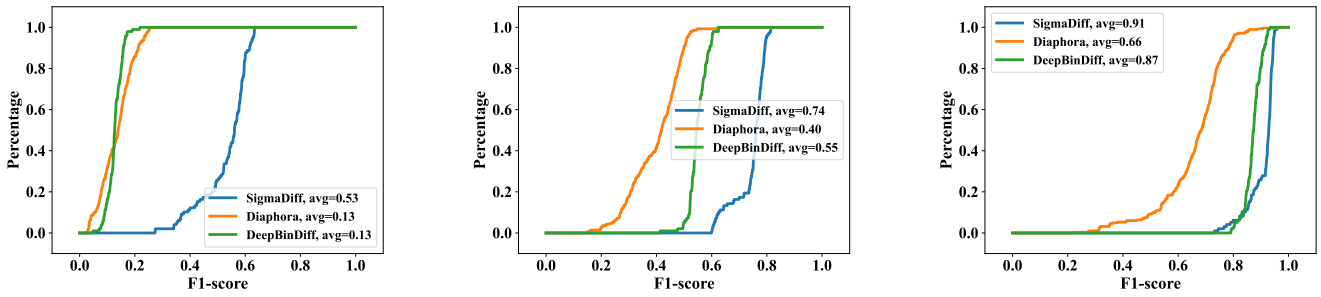
As shown in Table VI, SIGMADIFF outperforms Diaphora and DeepBinDiff in all the cases by large margins. On Gmp and Putty, the advantages of SIGMADIFF are relatively smaller because some large functions in Putty and Gmp exhibit significant differences, which are hard to match for SIGMADIFF and result in lower average token-level accuracy. Besides, the accuracy is further compromised due to the difficulty in matching functions. But overall, these experiments show that SIGMADIFF performs better pseudocode diffing on binaries built by different compilers, and it is able to generalize to unseen compilers, considering the model is pre-trained *only* on binaries compiled with GCC.

4) Cross-architecture Diffing:

We compare x86-64 binaries with ARM binaries. The dataset consists of all the binaries from Coreutils 8.1, Diffutils 3.6, Findutils 4.6, Gmp 6.2.1, and Putty 0.76 built with the default optimization settings. The evaluation results are shown in Table VII. Since SIGMADIFF and Diaphora rely on different disassemblers, the quality of the final pseudocode diffing also depends on the capability of Ghidra and IDA Pro in disassembling different architectures' binaries. For similar reasons in cross-compiler diffing, the diffing results in Gmp and Putty

TABLE V: Cross-version Pseudocode Diffing Results. Si: SIGMADIFF, De: DeepBinDiff, Di: Diaphora

		Recall			Precision			F1		
		Si	De	Di	Si	De	Di	Si	De	Di
Coreutils	v5.93 - v8.1	**0.624**	0.589	0.438	**0.770**	0.743	0.759	**0.687**	0.654	0.549
	v6.4 - v8.1	**0.689**	0.631	0.551	**0.793**	0.767	0.643	**0.735**	0.691	0.592
	Average	**0.656**	0.610	0.494	**0.781**	0.755	0.701	**0.711**	0.673	0.571
Diffutils	v2.8 - v3.6	**0.694**	0.660	0.348	**0.848**	0.806	0.406	**0.763**	0.725	0.375
	v3.4 - v3.6	**0.928**	0.909	0.788	0.957	**0.960**	0.844	**0.942**	0.934	0.815
	Average	**0.811**	0.784	0.568	**0.903**	0.883	0.625	**0.853**	0.829	0.595
Findutils	v4.233 - v4.6	**0.685**	0.579	0.366	**0.825**	0.758	0.773	**0.748**	0.655	0.487
	v4.41 - v4.6	**0.769**	0.690	0.499	**0.868**	0.847	0.786	**0.814**	0.759	0.609
	Average	**0.727**	0.635	0.433	**0.847**	0.803	0.779	**0.781**	0.707	0.548
Gmp	v6.0.0 - v6.2.1	**0.815**	N/A	0.691	0.871	N/A	**0.892**	**0.842**	N/A	0.779
	v6.1.1 - v6.2.1	**0.858**	N/A	0.771	0.894	N/A	**0.920**	**0.876**	N/A	0.839
	Average	**0.836**	N/A	0.731	0.882	N/A	**0.906**	**0.859**	N/A	0.809
Putty	v0.75 - v0.77	**0.781**	N/A	0.741	0.899	N/A	**0.897**	**0.836**	N/A	0.812
	v0.76 - v0.77	**0.798**	N/A	0.732	**0.908**	N/A	0.881	**0.849**	N/A	0.800
	Average	**0.789**	N/A	0.737	**0.904**	N/A	0.889	**0.842**	N/A	0.806



(a) O0 vs. O3 (b) O1 vs. O3 (c) O2 vs. O3

Fig. 6: Cross-optimization-level Pseudocode Diffing F1-score CDF (merging results of Coreutils, Diffutils, and Findutils)

TABLE VI: Cross-compiler Pseudocode Diffing Results (Clang vs. GCC). Si: SIGMADIFF, De: DeepBinDiff, Di: Diaphora

	Recall			Precision			F1		
	Si	De	Di	Si	De	Di	Si	De	Di
Coreutils 8.1	**0.595**	0.203	0.262	**0.807**	0.482	0.720	**0.681**	0.285	0.382
Diffutils 3.6	0.295	0.187	0.029	**0.574**	0.445	0.535	**0.390**	0.263	0.055
Findutils 4.6	0.363	0.202	0.051	**0.619**	0.458	0.597	**0.457**	0.279	0.094
Gmp 6.2.1	**0.393**	N/A	0.227	0.597	N/A	**0.821**	**0.474**	N/A	0.356
Putty 0.76	**0.273**	N/A	0.095	0.507	N/A	**0.610**	**0.354**	N/A	0.164

TABLE VII: Cross-architecture Pseudocode Diffing Results (ARM vs. x86-64). Si: SIGMADIFF, Di: Diaphora

	Recall		Precision		F1	
	Si	Di	Si	Di	Si	Di
Coreutils 8.1	**0.720**	0.429	0.725	**0.954**	**0.723**	0.586
Diffutils 3.6	**0.751**	0.003	**0.754**	0.674	**0.752**	0.006
Findutils 4.6	**0.620**	0.003	**0.642**	0.521	**0.631**	0.006
Gmp 6.2.1	**0.362**	0.165	0.386	**0.827**	**0.373**	0.275
Putty 0.76	**0.143**	0.038	0.436	**0.591**	**0.216**	0.072

are not as good as in the other programs. However, in general, we still observe that SIGMADIFF significantly outperforms Diaphora in the cross-architecture pseudocode diffing task.

5) C++ Programs Testing:

To evaluate the influence of indirect calls, we further evaluate SIGMADIFF on five C++ programs of different sizes. Specifically, we select `Stockfish` [20] from the Phoronix benchmark [17], three programs from `Thrift` [7], and one binary from `Xerces-c` [8]. The latter two are Apache Software Foundation [6] projects.

The evaluation results are listed in Table VIII. We list the size of the stripped program, number of indirect calls, recall, precision, and F1-score for each task. We approximate the number of indirect calls by counting the number of `CALLIND` opcode in Ghidra IR (`CALLIND` stands for indirect calls). SIGMADIFF outperforms DeepBinDiff and Diaphora on four out of five programs, while Diaphora performs better than SIGMADIFF in terms of F1-scores on `libthriftqt5.so`. This shows that SIGMADIFF is better at matching programs with larger differences, while Diaphora's heuristic-based function matching and string-based diffing achieve good results when two versions are close. DeepBinDiff relies on control flow graph and is significantly influenced by the indirect calls.

C. Efficiency

We evaluate the efficiency of the following four steps: pre-processing, training node selection, pre-training, and fine-tuning.

Pre-processing Time. On average, it takes 12.8s to pre-process

TABLE VIII: Diffing Results of C++ Programs. Si: SIGMADIFF, De: DeepBinDiff, Di: Diaphora

	Name	Size	#Indirect Calls	Recall			Precision			F1		
				Si	De	Di	Si	De	Di	Si	De	Di
Stockfish 14 vs. 15	stockfish	21.5M	88	**0.827**	N/A	0.760	0.939	N/A	**0.980**	**0.879**	N/A	0.856
Xerces-c 3.0.0 vs. 3.2.4	libxerces-c.so	3.6M	3,733	0.530	N/A	**0.533**	**0.840**	N/A	0.812	**0.650**	N/A	0.643
Thrift 0.13.0 vs. 0.17.0	libthrift.so	797K	1,435	**0.871**	N/A	0.867	**0.894**	N/A	0.784	**0.882**	N/A	0.824
	libthriftz.so	166K	152	**0.845**	0.641	0.831	**0.860**	**0.898**	0.786	**0.852**	0.748	0.808
	libthriftqt5.so	64K	124	0.927	0.565	**0.997**	0.927	0.942	**0.991**	0.927	0.707	**0.994**

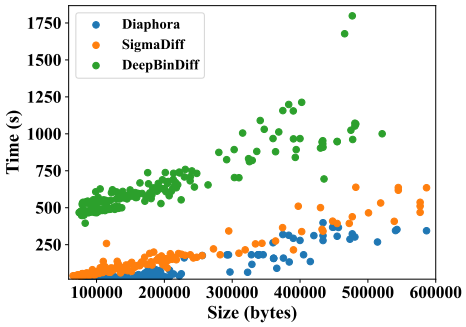


Fig. 7: Binary Size vs. Runtime of different models

one binary in our dataset. The pre-processing time is linear to the size of the binary.

Training Nodes Selection Time. SIGMADIFF takes 1.5s on average to perform the function matching and select the training nodes.

Pre-training Time. Pre-training is conducted only once during the whole evaluation. We pre-train the model on the `not` binary in the LLVM framework's implementation (v3.7.0 and v3.8.1). This program has no overlapping functions with our dataset. It takes 63 minutes to finish the pre-training on GPU.

Fine-tuning Time. SIGMADIFF takes 203s on average to perform the fine-tuning on GPU, which is the matching time since pre-training is not needed when conducting inference.

Figure 7 shows the runtime efficiency of SIGMADIFF, DeepBinDiff, and Diaphora versus different binary sizes. We admit that SIGMADIFF runs on GPU while other tools run on CPU. However, exploiting parallel computing power to solve the pseudocode diffing problem is one of our contributions. Thus such a comparison is still valid and meaningful. In general, with the help of GPU and our pre-training and fine-tuning schema, SIGMADIFF is quite efficient. It is much more efficient than DeepBinDiff, and it achieves similar efficiency to Diaphora when the binary size is small. Such efficiency indicates the practical application value of SIGMADIFF.

D. Ablation Study

Here we conduct ablation experiments to quantify the contribution of each key component in SIGMADIFF. Specifically, we set up five configurations:

- **SIGMADIFF-D**: SIGMADIFF that uses the doc2vec embeddings of the original IR text as the initial embeddings. And it leverages the 2-stage DGMC model.
- **SIGMADIFF-SD**: SIGMADIFF with lightweight symbolic analysis and the 2-stage DGMC model.
- **SIGMADIFF-STD**: SIGMADIFF with lightweight symbolic analysis, type constraints and the 2-stage DGMC model.
- **SIGMADIFF-STP**: SIGMADIFF with lightweight symbolic analysis, type constraints and pre-training, but only leverages the first stage of the DGMC model.
- **SIGMADIFF**: SIGMADIFF with lightweight symbolic analysis, type constraints, pre-training, and the 2-stage DGMC model.

TABLE IX: Recall, precision, and F1 of different configurations

Configurations	Recall	Precision	F1
SIGMADIFF-D	0.755	0.759	0.757
SIGMADIFF-SD	0.841	0.845	0.843
SIGMADIFF-STD	0.851	0.907	0.878
SIGMADIFF-STP	0.796	0.854	0.824
SIGMADIFF	0.868	0.916	0.891

We compare Diffutils v3.6 O2 and O3 binaries to evaluate the effectiveness of different configurations and the results are presented in Table IX. Comparing rows 1, 2, 3, and 5, we see that all key components positively impact the final results. Specifically, leveraging the lightweight symbolic analysis, type constraints, and pre-training technique improves the F1-score by 11.4%, 4.2%, and 1.5% respectively. Comparing rows 4 and 5, we find that the second stage of the DGMC model is significantly helpful. It improves the F1-score by 8.1%. Moreover, we evaluate the efficiency gain from pre-training. The average running time of the DGMC network for SIGMADIFF-STD and SIGMADIFF is 1,093s and 175s on this dataset, respectively.

E. Case Study

We further evaluate SIGMADIFF in security scenarios by conducting two case studies and comparing with BinDiff and Diaphora. In the first case study, we conduct patch detection on three open-source libraries where the ground truth is available, while in the second case study, we use a closed-source video conferencing application.

1) Patch Detection on Open-source Libraries: We first conduct a case study on eleven CVEs of `FFmpeg`, `libjpeg-turbo`, and `OpenSSL`. These CVEs are extracted from the CVE database (https://cve.mitre.org/), as shown in Table X. For each library, we select an older version with related CVEs and a more recent version in which the CVEs are patched. While the old version was only generated by GCC, we created two variants for the new version using two compilers (GCC and Clang). Both versions are compared using the default optimization settings. We compare each pair of the old and new versions using BinDiff, Diaphora, and SIGMADIFF, and calculate the recall score for detecting the patches in the new version. The ground truth of the patches (location of the patches) is obtained from the debug symbols.

The list of patches detected on each library is presented in Table X. We collect the statistics on all modification places and present the portion of detected diffs for each CVE. As demonstrated, when both of the two versions are compiled by GCC, all three tools can accurately locate most of the patches, while SIGMADIFF's performance is substantially better than the two baselines when the new versions are compiled by Clang and the old versions are compiled by GCC. This case study demonstrates that SIGMADIFF can pinpoint the patches even when there are compiler-introduced changes in the binary, which is useful given the difficulty of replicating identical building environments in practice.

TABLE X: List of CVE patches detected on open-source libraries. Si: SIGMADIFF, Di: Diaphora, Bi: BinDiff

Library	CVEs	GCC vs. GCC			GCC vs. Clang		
		Si	Di	Bi	Si	Di	Bi
FFmpeg	CVE-2020-22017	**1/1**	**1/1**	**1/1**	0/1	0/1	0/1
	CVE-2020-22024	**1/1**	**1/1**	**1/1**	**1/1**	0/1	0/1
	CVE-2020-22030	0/1	0/1	0/1	0/1	0/1	0/1
	CVE-2020-22034	**2/2**	1/2	1/2	0/2	0/2	0/2
	CVE-2020-22035	**2/2**	1/2	**2/2**	**2/2**	0/2	0/2
	CVE-2020-22040	**2/3**	1/3	**2/3**	1/3	0/3	0/3
libjpeg-turbo	CVE-2020-13790	**1/1**	**1/1**	**1/1**	**1/1**	0/1	0/1
	CVE-2021-46822	**1/2**	**1/2**	**1/2**	0/2	0/2	0/2
OpenSSL	CVE-2021-3712	**1/1**	0/1	**1/1**	**1/1**	0/1	**1/1**
	CVE-2021-3711	**1/2**	0/2	**1/2**	**1/2**	0/2	0/2
	CVE-2021-3449	**1/1**	0/1	0/1	**1/1**	0/1	**1/1**

2) Vulnerability Analysis on a Closed-source Application: We further conduct a case study on Zoom[4] [21], a widely-used video conferencing application to demonstrate how SIGMADIFF can help with vulnerability/patch analysis for real-world software. We investigate both the Windows (v5.9.7.3931) and Linux (v5.9.6.2225) versions and identify the shared and static libraries they use. Specifically, we have identified five open-source libraries and their versions based on the names of these binaries and strings embedded in these binaries. `OpenSSL`, `SQLite`, and `resiprocate` are statically linked into the main executable of the Linux version. `libjpeg-turbo` and `FFmpeg` are shared libraries included in the Windows version. The majority of libraries and binaries are over 1 MB, with the main executable being over 80 MB. And we utilize the iterative algorithm from §V-C to run the experiments, which takes around 12 hours in total.

To find known vulnerabilities (CVEs) in each of these identified dynamic and static libraries, we first build a binary (with debug symbols) for its latest version, locate the patches (functions and exact patch lines) for the CVEs that may exist in the identified library version according to the CVE database, and then perform diffing between the binary from Zoom and the corresponding binary of the latest version. For each of the static libraries, we directly match the main executable with the latest library. We evaluate SIGMADIFF, Diaphora, and BinDiff. DeepBinDiff was too slow to process these binaries.

Table XI lists the evaluation results. In total, with the help of SIGMADIFF, we are able to identify thirteen vulnerabilities in these five libraries in both the Linux and Windows versions of Zoom. SIGMADIFF can precisely pinpoint eight of these vulnerabilities at the token level. It means that in the pseudocode-level diffing results, the tokens related to the patch are precisely identified as insertion, deletion, or update, whereas the surrounding tokens are identified as matches. For the remaining five vulnerabilities, SIGMADIFF is able to locate the vulnerable functions. With some manual investigation, we are able to confirm the located functions were indeed vulnerable.

[4]We contacted the company, had a formal disclosure procedure, and got their consent to disclose all the vulnerability information mentioned in this paper. The developers have confirmed all the listed vulnerabilities.

TABLE XI: Diffing results for Zoom. Si: SIGMADIFF, Di: Diaphora, Bi: BinDiff

Library	Ver.	Confirmed CVEs	Function-lvl			Token-lvl		
			Si	Di	Bi	Si	Di	Bi
OpenSSL	1.1.1k	CVE-2023-0464	✓		✓	✓		
		CVE-2023-0215	✓					
		CVE-2022-4450	✓	✓		✓		
		CVE-2022-0778	✓	✓	✓	✓		✓
		CVE-2021-3712	✓	✓		✓		
		CVE-2021-3711	✓			✓		
SQLite	3.33.0	CVE-2022-35737	✓					
		CVE-2021-20227	✓					
resiprocate	1.11	CVE-2021-3672	✓	✓				
		CVE-2017-9454	✓	✓				
libjpeg-turbo	2.0.4	CVE-2020-13790	✓	✓	✓	✓	✓	✓
FFmpeg	4.2.3	CVE-2021-38291	✓	✓	✓	✓	✓	✓
		CVE-2020-22037	✓	✓	✓	✓		

In comparison, BinDiff can precisely spot three vulnerabilities at the basic block level and locate another two vulnerable functions. However, it is far more difficult for human analysts to examine the disassembly code than pseudocode.

Diaphora is able to pinpoint two vulnerabilities at the token level and locate eight vulnerable functions. But its diffing results are not as precise as the ones provided by SIGMADIFF. For example, in CVE-2021-3712, the vulnerable function, `EC_GROUP_new_from_ecparameters`, has over 300 lines of code. Due to information loss during compilation, decompilers have to infer high-level features, making variable names, memory access patterns, and control constructs susceptible to small changes in the assembly code. With a function of this size, the small changes soon cascade to a point that a simple string diffing algorithm cannot deal with. If we diff the whole function using Diaphora, no meaningful result is produced: almost the entire function is removed and then reinserted. Even if we focus on the code snippet surrounding the patch, as Figure 8 shows, the diffing result's quality is still considered low. String-based diffing is not suitable for this task. As a comparison, SIGMADIFF can find the exact change regardless of variable name or code structure changes.

From this case study, we can see that SIGMADIFF is able to discover significantly more vulnerabilities and produce more precise diffing results to ease manual investigation. We also verified using SIGMADIFF that the latest version (v5.14.2.2046) of Zoom has fixed all the CVEs listed in Table XI except CVE-2023-0464.

The results of this case study show that even famous commercial software may carry several vulnerable libraries, and some vulnerabilities have existed for almost five years. However, this is hard to avoid since it could be difficult to thoroughly audit all the third-party libraries and their dependencies, especially for large and complex projects, which signifies the necessity and practical value of SIGMADIFF.

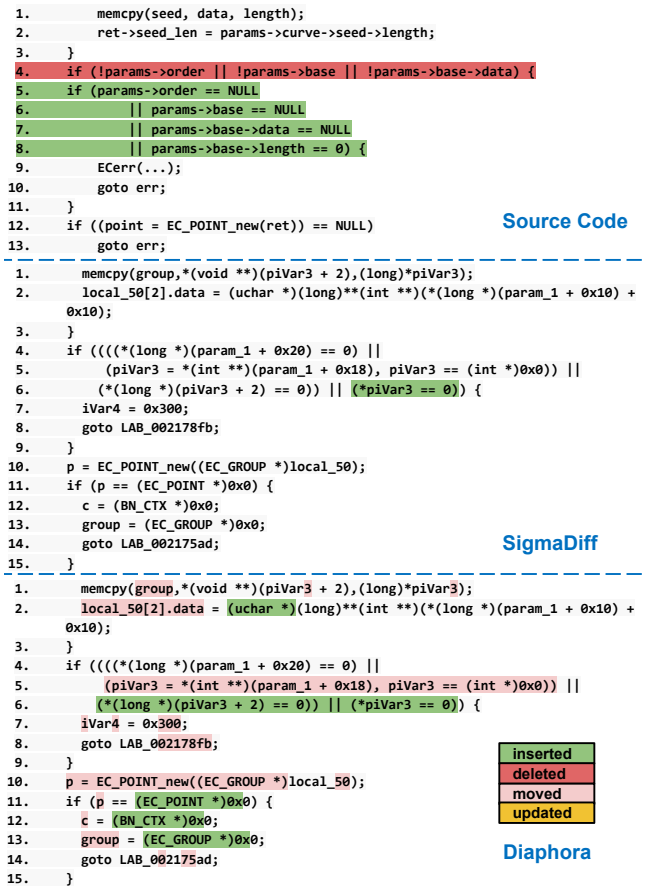


Fig. 8: Diffing result of CVE-2021-3712

VII. Related Work

A. Binary Diffing

Static Approaches. Some static techniques such as BinClone [42] and k-gram [64] leverage opcode and operand information to quantitatively measure the similarity of two binaries. Diaphora [9] uses multiple static diffing heuristics. Bindiff [22] and Binslayer [27] apply graph matching algorithms on CFGs. disovRE [40] optimizes runtime performance by choosing lightweight syntax features. Tracelet [36] converts CFG into *tracelets* that are essentially paths with fixed length and perform matching. Esh [33], GitZ [34] and FirmUp [35] decompose functions to *strands* by performing data-flow analysis. BinGo [29] inlines libraries into functions, extracts partial traces from CFG for matching. XMATCH [43] extracts conditional formulas for specific actions to conduct the code search. However, SigmaDiff generates formulas for all tokens, which is more complex.

Some other static approaches convert graphs into embeddings and transform graph matching problem into embedding similarity calculations. Genius [44] encodes attributed CFG into embeddings and leverages Locality Sensitive Hashing (LSH) for scalable online search. Gemini [76] further improves the embedding generation by using deep neural networks. InnerEye [84], Asm2Vec [37] and DeepBinDiff [38] utilize NLP techniques to learn instruction semantics to boost similarity calculation. αDiff [59] leverages CNN to generate function embeddings to avoid manually-crafted features.

Some existing approaches (e.g., XLIR [48]) leverage IR, but their purpose is for similarity detection or binary-source code matching. However, SigmaDiff is the first to conduct fine-grained binary diffing at the IR level.

Dynamic Approaches. In contrast to static approaches, which may be thwarted by techniques such as code obfuscations, research has been done to perform binary diffing via dynamic approaches. Blanket execution [39] executes function with the same input, monitors the behaviors, and performs comparison. BinSim [61] generates system call sliced segments, and checks their equivalence with symbolic execution and constraint solving. By nature, dynamic approaches are resistant to code obfuscation, but suffer from poor code coverage.

B. Patch-presence Test

Patch-presence tools search for a patch in an unknown target. Fiber [79] chooses the most appropriate parts of a patch to create binary signatures that accurately represent the source-level patch. It is evaluated on Android kernel images. PDiff [53] further provides improvements for downstream kernels by generating semantic patch summaries. BScout [32] associates raw Java bytecode instructions with Java source code lines, and compares changes with pre-patch/post-patch source code. These tools assume that function names are already available or easy to recover while SigmaDiff does not. Besides, SigmaDiff could also be applied in other tasks such as plagiarism detection and lineage analysis.

C. Graph Learning Models

Traditional graph matching algorithms such as Hungarian algorithm [56] try to find maximum-weight matchings in bipartite graphs. Cho et al. [30] learned a graph model based on representations with histogram-based attributes for nodes and edges. Bayati et al. [26] developed a message passing algorithm for network alignment problem. In general, these techniques treat graph matching as an assignment problem on graphs, which is NP-hard, and do not scale very well.

Graph Representation Learning. Multiple recent methods leverage deep learning to learn graph embeddings, which can facilitate graph matching. Works such as DeepWalk [66] and node2vec [47] encode graph information based on random walks. LINE [72] combines two encoder-decoder objectives that optimize graph proximity. DNGR [28] and SDNE [74] leverage autoencoders to compress node's local structural information. Some other techniques such as GCN [55] and GraphSAGE [49] employ convolutional networks or LSTM as aggregators to encode graph information. TADW [78] considers feature vectors for each node and perform matrix factorization to generate graph embeddings.

Deep Graph Matching. GSimCNN [24] uses GCNs to approximate the graph edit distance between two graphs. GraphUIL [81] proposes a deep graph model based on global and local network topology preservation. CMPNN [82] uses GNNs to transform coordinates of feature points into local features and discover the optimal alignment. DGMC [46] applied in SigmaDiff uses localized node embeddings computed by a GNN to obtain an initial ranking, and employs synchronous message passing networks to reach a matching consensus. It strikes a nice balance between scalability and global node correspondences consistency.

VIII. Discussion

In this section, we discuss the limitations and unsolved challenges of SigmaDiff.

PDG changes. Since our approach conducts the matching on two PDGs, the effectiveness of our approach could be limited by the code transformations that influence the stability of PDG, such as loop unrolling, function inlining, and code obfuscation. Note that our inter-procedural PDG can only partially solve the function inlining problem (mainly with simple cases), because the structure of the graphs are still different considering we have added extra nodes to the graph. Moreover, there are cases when callees are only partially inlined for some paths. These PDG changes will adversely influence our matching accuracy.

Lightweight symbolic analysis. Our lightweight symbolic analysis is robust in general and can address some of the code transformations caused by function inlining since it is inter-procedural. However, since our primary goal is to extract semantic information efficiently, we sacrifice soundness to some extent. Besides, the generated expressions are not helpful for matching SSE instructions with other instructions, since they are completely different at IR level.

One-to-many matching. SigmaDiff can only perform one-to-one matching of the pseudocode tokens. However, in reality, a token may need to be matched with multiple tokens. We plan to systematically address this in our future work.

Indirect calls/jumps. We ignore indirect calls/jumps when generating the callgraph and IPDG, which will influence the accuracy of diffing (evaluated in §VI-B5). Nevertheless, this limitation exists for all diffing tools. In our future work, we aim to systematically tackle this issue.

Scalability. Our current iterative algorithm can deal with large binaries that cannot fit in the GPU memory at once. However, it is inefficient as it matches one function pair at a time. A better iterative algorithm should better utilize the GPU memory and process larger subgraphs per iteration. We leave the improvement of the iterative algorithm as future work.

IX. Conclusion

In this paper, we have proposed a model called SigmaDiff for pseudocode diffing. SigmaDiff leverages binary program analysis techniques and deep learning to achieve accurate and scalable pseudocode diffing. It consists of three stages: pre-processing, diffing, and post-processing. Extensive experiments have been conducted to compare SigmaDiff with the state-of-the-art deep learning-based model (DeepBinDiff) and the open-source solution (Diaphora). Experimental results show that SigmaDiff significantly outperforms these models in terms of accuracy and efficiency. Large-scale case studies also signify the necessity and practical value of SigmaDiff.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported by NSF under grant No. 1719175 and 2022 Amazon Research Award.

References

[1] Automated struct identification with ghidra. https://blog.grimm-co.com/2020/11/automated-struct-identification-with.html, 2020.

[2] Cve-2020-13790 details. https://nvd.nist.gov/vuln/detail/CVE-2020-13790, 2020.

[3] Gnu coreutils. https://www.gnu.org/software/coreutils/, 2020.

[4] Symbolic value set analysis. https://github.com/penhoi/ghidra-decompiler/wiki/Symbolic-Value-Set-Analysis, 2020.

[5] addr2line. https://sourceware.org/binutils/docs/binutils/addr2line.html, 2022.

[6] Apache software foundation. https://www.apache.org/, 2022.

[7] Apache thrift. https://thrift.apache.org/, 2022.

[8] The apache xerces project. https://xerces.apache.org/, 2022.

[9] Diaphora. http://diaphora.re/, 2022.

[10] Gnu diffutils. https://www.gnu.org/software/diffutils/, 2022.

[11] Gnu findutils. https://www.gnu.org/software/findutils/, 2022.

[12] The gnu mp bignum library. https://gmplib.org/, 2022.

[13] Gumtree heuristics. https://tinyurl.com/53kpf735, 2022.

[14] Ida disassembler and debugger. https://www.hex-rays.com/products/ida/, 2022.

[15] libjpeg-turbo. https://libjpeg-turbo.org/, 2022.

[16] Llvm language reference manual. https://llvm.org/docs/LangRef.html, 2022.

[17] Phoronix. https://openbenchmarking.org/tests, 2022.

[18] Putty: a free ssh and telnet client. https://www.chiark.greenend.org.uk/~sgtatham/putty/, 2022.

[19] Retdec, a retargetable machine-code decompiler based on llvm. https://github.com/avast/retdec, 2022.

[20] Stockfish. https://stockfishchess.org/, 2022.

[21] Zoom. https://zoom.us/, 2022.

[22] Zynamics bindiff. https://www.zynamics.com/bindiff.html, 2022.

[23] National Security Agency. Ghidra reverse engineering tool. https://ghidra-sre.org/, 2022.

[24] Yunsheng Bai, Hao Ding, Yizhou Sun, and Wei Wang. Convolutional set matching for graph similarity. *arXiv preprint arXiv:1810.10866*, 2018.

[25] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):1–84, 2010.

[26] Mohsen Bayati, David F Gleich, Amin Saberi, and Ying Wang. Message-passing algorithms for sparse network alignment. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 7(1):1–31, 2013.

[27] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–10, 2013.

[28] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

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

[30] Minsu Cho, Karteek Alahari, and Jean Ponce. Learning graphs to match. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 25–32, 2013.

[31] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[32] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. BScout: Direct whole patch presence test for java executables. In *29th*

USENIX Security Symposium (USENIX Security 20), pages 1147–1164, August 2020.

[33] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *ACM SIGPLAN Notices*, 51(6):266–280, 2016.

[34] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–94, 2017.

[35] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 53(2):392–404, 2018.

[36] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *Acm Sigplan Notices*, 49(6):349–360, 2014.

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

[38] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*, 2020.

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

[40] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[41] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[42] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

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

[44] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[45] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[46] Matthias Fey, Jan E. Lenssen, Christopher Morris, Jonathan Masci, and Nils M. Kriege. Deep graph matching consensus. In *International Conference on Learning Representations*, 2020.

[47] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[48] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 601–612. IEEE, 2022.

[49] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[50] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Computing Surveys (CSUR)*, 54(3):1–38, 2021.

[51] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, 1988.

[52] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. Cldiff: generating concise linked code differences. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 679–690. IEEE, 2018.

[53] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1149–1163, 2020.

[54] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. iclr. 2015. *arXiv preprint arXiv:1412.6980*, 9, 2015.

[55] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[56] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[57] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[58] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, page 3236–3251, 2021.

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

[60] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 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*, pages 389–400, 2014.

[61] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 253–270, 2017.

[62] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security and Privacy Conference*, pages 416–430. Springer, 2015.

[63] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[64] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 314–318, 2005.

[65] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. Nil: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 830–841, 2021.

[66] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[67] Ignacio Rocco, Mircea Cimpoi, Relja Arandjelović, Akihiko Torii, Tomas Pajdla, and Josef Sivic. Neighbourhood consensus networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 1658–1669, 2018.

[68] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128, 2009.

[69] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences . . . , 1978.

[70] Olin Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, 1988.

[71] Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices

and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

[72] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015.

[73] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[74] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234, 2016.

[75] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 319–330. IEEE, 2017.

[76] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 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*, pages 363–376, 2017.

[77] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE, 2017.

[78] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Y Chang. Network representation learning with rich text information. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 2111–2117, 2015.

[79] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, August 2018.

[80] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[81] Wen Zhang, Kai Shu, Huan Liu, and Yalin Wang. Graph neural networks for user identity linkage. *arXiv preprint arXiv:1903.02174*, 2019.

[82] Zhen Zhang and Wee Sun Lee. Deep graphical feature learning for the feature matching problem. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5087–5096, 2019.

[83] Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 141–151, 2018.

[84] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS (2019)*, 2019.

APPENDIX A
HYPERPARAMETERS SELECTION

Embedding Sizes. We test five sets of embedding sizes, where the first embedding size is the dimension of the hidden states of Ψ_{θ_1}, and the second embedding size is the dimension of the hidden states of Ψ_{θ_2}. The initial node embeddings generated by doc2vec have the same dimension as the first embedding size. This experiment is conducted on the `cmp` binary in Diffutils 3.6 O2 and O3. Table XII shows the recall, precision and F1-score when SIGMADIFF is configured with different embedding sizes. We can observe that the performance of SIGMADIFF increases as the embedding sizes grow. In our experiments, we choose the embedding sizes 128 and 32 based on our hardware capacity.

Number of Hops. Figure 9 shows the F1-score during training when the number of hops (see §V-C) in Ψ_{θ_1} is set to different values. The model is trained on the `cmp` binary in Diffutils 3.6 O2 and O3 for 800 epochs. As shown in the figure, adding more hops is helpful for speeding up the training and improving the final accuracy. In this paper, we set hops to be 3 considering both accuracy and efficiency.

TABLE XII: Embedding sizes' influences

Embedding Sizes		Recall	Precision	F1
GNN1	GNN2			
32	16	0.518	0.705	0.597
64	16	0.827	0.893	0.859
128	32	0.872	0.922	0.896
256	32	0.881	0.931	0.905
512	64	0.915	0.952	0.933

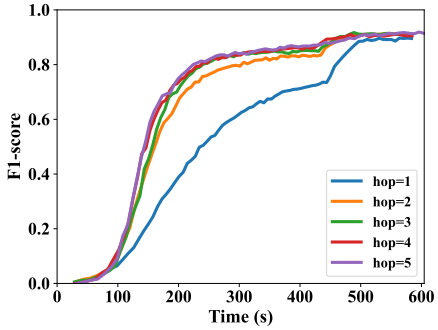


Fig. 9: F1-scores with different *Number of hops* values

APPENDIX B
EXAMPLE OF MERGE AND SOUNDNESS ANALYSIS

Example. As mentioned in §IV-B, the `merge` operation is helpful for handling branches/loops. For example, the value of variable `i` in Figure 10 is merged from two paths: (1) the initialization at the beginning, (2) the $i \mathrel{+}= 4$ at the end of each iteration. During analysis, the symbolic expression of i is 0 at the beginning. Then i is incremented by 4, and the algorithm merges the old symbolic expression of i with the new one 4. The merged result is $\emptyset$, since $0, 4 \notin \{ARGn, Str, Symbol\}$ and $\texttt{sources}(0) = \texttt{sources}(4) = \emptyset$. When i is equal to 8, the symbolic expression of i is still $\emptyset$ due to the same reason. Therefore a fixed point is reached and we exit the for-loop. Compared to value-set analysis that estimates a range of numeric values, our approach is more focused on how a value is calculated. Therefore, we choose to only keep the sources of the expression to simplify the merge operation but still record a certain amount of semantic information.

```
1.   for( int i = 0; i < 16; i += 4 ) {
2.       *(param_1 + i) = param_2;
3.   }
4.   uVar1 = *(param_1 + 4)
```

Fig. 10: A simple pseudocode snippet

Soundness Analysis. In most cases, the symbolic expressions over approximate concrete values. But there are unsound cases due to the fact that we cannot compare the equivalence of two symbolic expressions. For example, in Figure 10, $param\_1+i$ and $param\_1 + 4$ could point to the same memory location. The first one is dereferenced in the loop to store the value $param\_2$ while the second one is dereferenced later to load the value. We will consider the loaded value different from the stored value since the symbolic expressions of the two

addresses are not equal (i.e., $\{ARG1\}$ and $ARG1 + 4$). As a result, we will lose track of the stored value $param\_2$.

APPENDIX C
FUNCTION FEATURES AND FUNCTION MATCHING RESULTS

We explain the extracted function features as follows. Note that our purpose of function matching is to help locate the training nodes of the DGMC model with high quality.

- ***Return values.*** We collect the symbolic values of the return values from each of the return IR within the function.
- ***Side effects.*** The side effects happen when the function writes to memory outside the current stack frame. The side effect feature set includes all the memory address and the values that are written to the memory. For instance, in the example shown in Figure 4, the side effect feature set would be {"ARG1+8:ARG1+20"} ($ID = 7$).
- ***Loads.*** We also consider the memory loads whose addresses are related to the function parameters. We collect the addresses of these loads as the load feature set.
- ***Strings and library calls.*** We only consider the function names of the library calls and ignore the parameters.
- ***Function parameter types.*** Function parameter types include primitive types such as `boolean`, `int`, recovered by the decompiler, and also pointers of data structures.

To calculate the similarity score between two functions, we first calculate a Jaccard similarity for the *Return values*, *Side effects*, *Loads*, *Strings and library calls* feature sets respectively. Then we calculate the similarity score of *Function parameter types*. In this step, primitive types are directly counted. For pointer types, we calculate the Jaccard similarity of the data structure fields they point to. Then we calculate the final similarity score between the two functions by using a multi-layer perceptron (MLP), considering different features should have different impacts on the final similarity. Specifically, we train the MLP weights using the binary functions from LLVM (v3.7.0 and v3.8.1) framework. Function pairs with the same name are labeled as 1 and function pairs with different names are sampled randomly and labeled as 0.

After obtaining the function similarity scores through MLP, we leverage a 2-hop greedy matching algorithm to conduct a function-level matching. The algorithm is the same as the *k*-hop algorithm in DeepBinDiff [38]. *k* is set to be two to handle function inlining, while in DeepBinDiff it is four due to block reordering. The main idea is to match along the call graph starting from the most similar functions. Specifically, we locate new matching functions by comparing the 2-hop neighbors to functions that have already been matched so that the context provided by the call graph can be leveraged.

TABLE XIII: Function Matching Results

	Precision		Recall	
	Avg.	Stdev.	Avg.	Stdev.
O0 vs. O3	0.775	0.043	0.707	0.038
O1 vs. O3	0.884	0.037	0.841	0.034
O2 vs. O3	0.934	0.036	0.879	0.030

Function matching is only an intermediate step of SIGMADIFF whose purpose is to locate the training nodes for the DGMC model. However, in order to show the robustness of the extracted features, and the performance of our function-level matching module (see §V-B), we report precision and recall scores of this step (in the cross-optimization-level diffing task) in Table XIII. The dataset includes all the versions of the binaries in Coreutils, Diffutils, and Findutils. As shown in the table, SIGMADIFF has reasonably good function matching results which are further leveraged in the following training nodes selection and pseudocode diffing stages.

APPENDIX D
NUMBER OF TRAINING NODES

To demonstrate the impact of the training nodes, we investigate the percentage of the training nodes relative to the total number of nodes in the Diffutils experiments. As shown in Table XIV, the percentage of training node pairs is related to the difficulty of diffing. For instance, there are more training nodes generated when comparing O2 and O3 than comparing O0 and O3. The percentage of training nodes in Clang vs. GCC and ARM vs. x86-64 is also notably low. This aligns with the final diffing accuracy.

TABLE XIV: The training node percentage in Diffutils experiments.

Configurations	Percentage	Configurations	Percentage
O0 vs. O3	12.8%	v3.4 vs. v3.6	36.2%
O1 vs. O3	22.7%	Clang vs. GCC	14.6%
O2 vs. O3	31.4%	ARM vs. x86-64	18.2%
v2.8 vs. v3.6	22.8%	-	-

APPENDIX E
COMPARISON WITH SOURCE CODE CLONE DETECTION

In this section, we compare SIGMADIFF with a state-of-the-art token-based source code clone detector, NIL [65]. Specifically, we run NIL on two sets of pseudocode functions and extract the token-level code clone detection results as the matching results to compare with SIGMADIFF. NIL is based on an N-gram representation of token sequences and an inverted index. By determining the longest shared subsequence between the token sequences, the similarity is measured. Table XV shows NIL's precision, recall, and F1-score on cross-version and cross-optimization-level diffing tasks on Diffutils.

TABLE XV: Comparison with NIL on Diffutils. Si: SIGMADIFF, Ni: NIL

	Recall		Precision		F1	
	Si	Ni	Si	Ni	Si	Ni
v2.8 vs. v3.6	**0.694**	0.072	**0.848**	0.718	**0.763**	0.126
v3.4 vs. v3.6	**0.928**	0.473	**0.957**	0.858	**0.942**	0.604
v2.8 O2 vs. O3	**0.868**	0.325	**0.916**	0.860	**0.891**	0.471

In general, NIL does not work well on pseudocode diffing tasks especially when the code changes are large. SIGMADIFF outperforms NIL by large margins in terms of F1-scores on all three tasks. NIL's performance drops significantly when the difference between two code sets increases. Since it relies on the longest shared subsequence, the matching accuracy is largely influenced when the token sequences change. It cannot perform the cross-opt-level diffing well due to the same reason. By looking at Table XV and Table V, we can observe that NIL performs even worse than Diaphora. These results show that source code clone detection (at token level) techniques cannot be leveraged in pseudocode diffing.

APPENDIX F
ARTIFACT APPENDIX

This artifact includes a release of our proposed pseudocode diffing prototype, SIGMADIFF. It also includes a sample script to show how we can use it to perform cross-optimization-level, cross-version, cross-architecture, and cross-compiler diffing among sets of binaries, which outputs the diffing results and evaluation results.

A. Description & Requirements

1) How to access: The full artifact is available on GitHub at the following URL: https://github.com/yijiufly/SigmaDiff/tree/v0.1. The DOI link is https://doi.org/10.5281/zenodo.8287857.

2) Hardware dependencies: A machine with NVIDIA GPU with at least 11GB of memory and compute capability of 5.2 or above is required. Here is a list of NVIDIA GPUs and their compute capabilities: https://developer.nvidia.com/cuda-gpus.

3) Software dependencies:

- Ubuntu 20.04 system
- CUDA 11.1
- Python 3.9.7
- Pytorch 1.9.1
- Gensim 4.0.1
- Scipy 1.10.1
- Torch-geometric 2.0.4
- Torch-scatter 2.0.9
- Torch-sparse 0.6.12
- Numpy 1.23.2
- Ghidra 9.2.2
  - need to import `json-simple-1.1.1.jar` to ghidra, see README for detailed instructions.
- Java 11

4) Benchmarks: Since the complete experiment in our evaluation section is lengthy, we only include the Coreutils dataset in this artifact. It is located in the `data/binaries` directory. We include three versions (v5.93, v6.4, v8.1) of Coreutils, and include different optimization levels (from O0 to O3) for version 5.93. We also include the clang version and arm version of Coreutils in this directory. The source code of Coreutils is located at `data/sources`, which is needed during the evaluation of the cross-version diffing.

The complete dataset of our evaluation can be downloaded from https://drive.google.com/drive/folders/1IimJi-03B4ljogtk4hli6B5G12MnpWJ-?usp=sharing.

B. Artifact Installation & Configuration

To prepare the environment for the evaluation, it is recommended to create a conda environment and install all the software dependencies listed in §F-A3.

Our release requires no installation. Detailed instructions on using and running the tool are included in the README file.

C. Major Claims

- (C1): SIGMADIFF outperforms the state-of-the-art systems in cross-version, cross-optimization-level, cross-compiler, and cross-architecture diffing tasks. This is proven by the experiment (E1) whose results are reported in Table V, Figure 6, Table VI, and Table VII respectively in the paper.
- (C2): SIGMADIFF is efficient with the help of GPU and the pre-training and fine-tuning schema. This is also proven by the experiments (E1) whose results are reported in Figure 7.

D. Evaluation

1) Experiment (E1): [2 hours for setting up the environment and 20-50 hours of execution time (depending on the GPU)]: This experiment performs cross-optimization-level, cross-version, cross-architecture, and cross-compiler diffing on the Coreutils binary set. We only provide a scaled-down version of our paper's evaluation but it is sufficient to confirm our conclusions since Coreutils is representative and it consists of binaries of different sizes. Our evaluation metrics include precision, recall, and F1-score. We also evaluate the efficiency of SIGMADIFF by comparing the execution time with the size of the binaries.

[How to] Run `./run.sh`. The detailed diffing output can be found in the `out` directory, the evaluation results (precision, recall, and F1-score) can be found in `out/finalresults.txt`, and the execution time results can be found in `out/time.txt`.

[Preparation] Before running `./run.sh`, the `json-simple-1.1.1.jar` needs to be imported to a Ghidra project (see README for detailed instructions). Then the Ghidra home directory and the Ghidra project name must be specified in `run.sh`.

[Execution] Run `./run.sh`.

[Results] The evaluation script `run.sh` will generate evaluation results in `out/finalresults.txt`. Each row shows the average precision, recall, and F1-score of our diffing results for each group of binaries. In total, it should generate 7 lines of results. These results should be compared with the corresponding rows in Table V, Table VI, Table VII in the paper and the detailed cross-optimization-level results in the artifact repository. It is possible there are variations caused by the DGMC model. In most cases, the variation should be around 0.05. The maximum variation observed is approximately 0.1. It is expected that the SIGMADIFF algorithm will achieve better results than the baselines in terms of F1-score.

The execution time results are in `out/time.txt`. Each row represents a pair of execution samples and shows the execution time as well as the average size of the samples. The data in each row corresponds to a dot depicted in Figure 7. It is expected that the dots fall in the orange range approximately in the figure if the GPU capability is similar to RTX 2080Ti. If the GPU has a lower capability than the RTX 2080Ti, it will result in a longer execution time.