Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection

Haojie He, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Xingwei Lin, *Ant Group;* Ziang Weng and Ruijie Zhao, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Shuitao Gan, *Laboratory for Advanced Computing and Intelligence Engineering;* Libo Chen, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Yuede Ji, *University of North Texas;* Jiashui Wang, *Ant Group;* Zhi Xue, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University*

https://www.usenix.org/conference/usenixsecurity24/presentation/he-haojie

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection

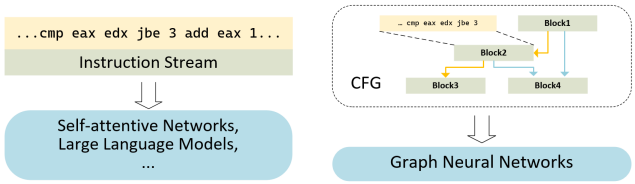
Haojie He[1], Xingwei Lin[2], Ziang Weng[1],
Ruijie Zhao[1], Shuitao Gan[3], Libo Chen✉[1], Yuede Ji[4], Jiashui Wang[2], and Zhi Xue[1]
[1]School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University [2]Ant Group
[3]Laboratory for Advanced Computing and Intelligence Engineering [4]University of North Texas
{sgfvamll, ziangweng, ruijiezhao, bob777, zxue}@sjtu.edu.cn,
{linyi.lxw, quhe}@antgroup.com, ganshuitao@gmail.com, yuede.ji@unt.edu

Abstract

Binary code similarity detection (BCSD) has garnered significant attention in recent years due to its crucial role in various binary code-related tasks, such as vulnerability search and software plagiarism detection. Currently, BCSD systems are typically based on either instruction streams or control flow graphs (CFGs). However, these approaches have limitations. Instruction stream-based approaches treat binary code as natural languages, overlooking well-defined semantic structures. CFG-based approaches exploit only the control flow structures, neglecting other essential aspects of code. Our key insight is that *unlike natural languages, binary code has well-defined semantic structures, including intra-instruction structures, inter-instruction relations (e.g., def-use, branches), and implicit conventions (e.g. calling conventions)*. Motivated by that, we carefully examine the *necessary* relations and structures required to express the *full* semantics and expose them directly to the deep neural network through a novel semantics-oriented graph representation. Furthermore, we propose a lightweight multi-head softmax aggregator to effectively and efficiently fuse multiple aspects of the binary code. Extensive experiments show that our method significantly outperforms the state-of-the-art (e.g., in the x64-XC retrieval experiment with a pool size of 10000, our method achieves a recall score of 184%, 220%, and 153% over Trex, GMN, and jTrans, respectively).

1 Introduction

Binary code similarity detection (BCSD) is a fundamental task that determines the semantic similarity between two binary functions. It serves as a crucial component in addressing various important challenges, including retrieving known vulnerable functions in third-party libraries or firmware [11, 15, 26, 29, 42], recovering library function symbols in statically linked binaries [9, 10, 27], detecting software plagiarism [25], detecting software license violations [17], and identifying malware code snippets [4]. Nevertheless, it is very challenging to develop general and efficient detection methods. On one hand, two semantically identical code snippets may exhibit entirely different syntax representations, particularly when a piece of source code is compiled into different instruction set architectures. On the other hand, two semantically different snippets can possess similar syntax representations. Thus, understanding the high-level semantic features of code is the key to effectively performing BCSD.



(a) Instruction Stream-based (b) CFG-based

Figure 1: Two lines of works on code semantics learning.

Existing works can generally be classified into two directions, i.e., instruction streams-based and control flow graph (CFG)-based, as illustrated in Figure 1.

Instruction streams-based methods treat instruction streams as if they were natural language sentences and consecutively introduce natural language processing (NLP) techniques, such as long short-term memory (LSTM) networks [45], self-attentive networks [29], large language models [1], and sophisticated pre-training techniques [33, 40, 43]. The state-of-the-art methods in this direction leverage specially designed pre-training tasks, such as jump target prediction [40] and block inside graph detection [43], to enable the deep neural network models to grasp code semantics. However, the pre-training process is expensive as it relies on large language models and large-scale datasets. In addition, the pre-training tasks can usually be solved reliably and fast by traditional program analysis algorithms. Therefore, instead of teaching models to solve these tasks based on a low-level representation, it may be more beneficial to recover these

✉Corresponding author.

well-defined semantic structures using traditional methods and present them to deep neural networks to learn higher-level semantics. Moreover, existing work focuses on learning only limited aspects of code, such as control flow structures or contextualized syntax probabilities.

CFG-based methods mainly leverage the graph neural networks (GNN) and typically take CFGs as the input, which are widely recognized as crucial features in analyzing binary code [2,10,11,14]. Recent works [18,20,26,28,43,44] in this direction combine NLP-based methods to learn basic block features with GNNs to capture control flow characteristics. Some of them have achieved state-of-the-art [26,44]. However, the reliance on NLP-based methods makes them suffer from the same predicament as the previous direction of work. Other works seek to exploit additional code semantics beyond the control flow, such as enhancing the CFG with inter-basic-block data flow edges [15] and incorporating data flow graphs (DFGs) [16]. These works have taken an involuntary step towards utilizing full semantic structures of code.

Overall, the stream representation with NLP-based methods is adopted by state-of-the-art works in both directions. Although the instruction streams and natural language sentences are similar in syntax, they differ implicitly:

(i) First, natural languages are ambiguous and weakly structured while the binary code has well-defined structures, semantics, and conventions. For instance, accurately parsing the syntactic dependency in natural language sentences is challenging, while it is easy in binary code.

(ii) Second, reordering instructions or moving instructions between basic blocks is feasible, which suggests that binary code should be represented in a more flexible representation rather than sequences. In fact, instructions are often reordered and moved around basic blocks by the compiler for optimization. However, there is no well-defined way to reorder words in natural languages while preserving the semantics.

(iii) Third, natural languages are designed for effectively exchanging information while binary code is designed to ease the machine execution. For instance, most 'words' in assemble languages are 'pronouns'. When instructions use registers or memory slots, they indeed use the value temporarily cached inside them. The exact registers or memory slots used to cache the value are semantics-independent.

These differences suggest that treating code as natural language is suboptimal. In light of this, we suggest the development of a binary code representation that is capable of (1) revealing intra-instruction structures that show how operands are used by operators, (2) revealing inter-instruction relations such as def-use, control flow, and necessary execution order, (3) excluding semantics-independent elements such as the registers used to temporarily cache data and unnecessary execution order restriction, and (4) encoding other implicit knowledge, e.g., calling conventions. (See §2 for detail.)

In this paper, we propose a novel binary code representation, named semantics-oriented graph (SOG), which possesses the aforementioned capabilities. Based on SOG, we adopt a GNN-based approach for semantics learning. While replacing the CFG with the SOG in the existing GNN-based framework offers certain advantages, it fails to unleash the full potential of the SOG. We further identify a short slab of the existing GNN-based framework: the graph aggregation module. SOG encodes multiple aspects of code. Considering that different aspects of the SOG should be useful to distinguish one function from different types of other functions, we propose a novel multi-head softmax aggregator to fuse them simultaneously. This aggregator is powerful yet lightweight.

By integrating the SOG and the multi-head softmax aggregator, we build an effective and efficient binary code similarity detection solution, HermesSim. HermesSim first lifts a binary function to the SOG representation and then utilizes the graph neural network to aggregate neighbor information for each node. After that, the multi-head softmax module is applied to generate a graph embedding vector. The similarity between functions is then approximated by the similarity between their embedding vectors. Besides, HermesSim adopts the margin-based pairwise loss [21] along with the distance-weighted negative sampling strategy [41] for training. It is noteworthy that HermesSim has two orders of magnitude fewer parameters than previous methods based on large language models.

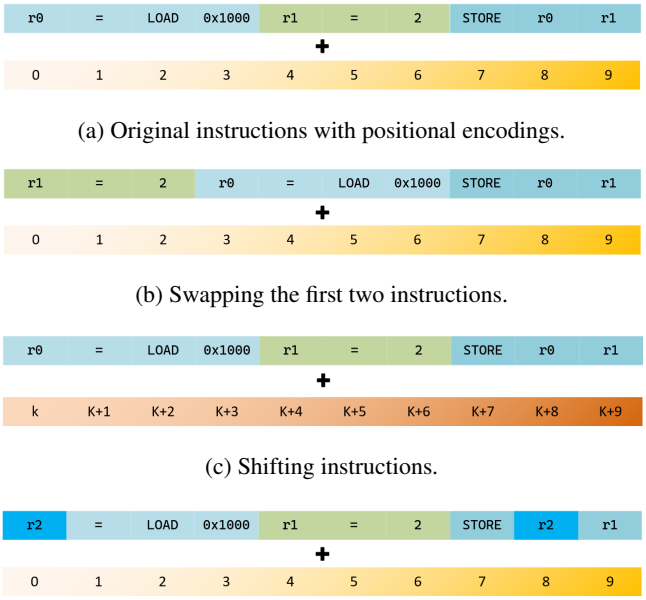
In summary, we make the following contributions:

- We propose a novel binary code representation named semantics-oriented graph (SOG) for BCSD, and we detail its construction. This representation not only reveals complete semantic structures of binary code but also discards semantically independent information. To the best of our knowledge, this is the first investigation in this direction.
- We design a novel multi-head softmax aggregator to properly aggregate multiple aspects of SOG. This module significantly enhances the performance of our system.
- We design and implement HermesSim, an effective yet efficient solution for binary code similarity detection.
- We perform extensive experiments and demonstrate the effectiveness of SOG over previous mainstream binary code representations. Moreover, we conduct both laboratory experiments and real-world 1-day vulnerability searches, establishing HermesSim's significantly superior performance over the state-of-the-art methods.

Open Source. All our artifacts are available at https://github.com/NSSL-SJTU/HermesSim.

2 Background and Motivation

This section aims to answer two questions: why do we propose the SOG, and what is it? We first discuss the additional semantics that the sequence representation imposes on NLP-based methods to learn (§2.2). Then we elaborate on the implicit structures of binary code (§2.3). Finally, we present an intuitive explanation of the SOG (§2.4).



(a) Original instructions with positional encodings.

(b) Swapping the first two instructions.

(c) Shifting instructions.

(d) Replacing the uses of the register r0 with r2.

Figure 2: Three semantically equivalent variants of a simple instruction stream snippet.

To avoid unnecessary complexity, we present ideas on a toy intermediate representation (IR) (§2.1). The ideas developed can be easily extended to real-world representations.

2.1 A Toy IR

The toy IR is mostly self-explanatory. As shown in Figure 3a, the toy IR includes four types of tokens, i.e. registers which are of the form `ri` (e.g. `r1`), instruction operators which appear as the first token of the instruction or the first token on the right of the equation symbol (e.g. `STORE`, `CALL`), integer literals and labels (e.g. `L1`). Labels mark the start of basic blocks and are used by branch instructions.

The semantics of the code snippet in Figure 3a is explained below. Labels `L1` at line 1, and `L2` at line 7 indicate the start of two basic blocks separately. The instruction `r0 = LOAD 0x1000` loads the value of the memory slot at address 0x1000 and places it into the register `r0`. Following that, `r1 = ADD r1, 2` stores the value of expression `r1 + 2` into the register `r1`. `STORE r0, r1` stores the value of `r1` into the memory address indicated by `r0`. The instruction `r3 = CALL Foo, r0` invokes the subroutine `Foo` with `r0` as the only argument and places the returned value into `r3`. `BR L2` jumps to the basic block marked by the label `L2` directly. Finally, `RET r3` returns the control to the caller with the return value set as the value stored in `r3`.

2.2 Semantically Equivalent Variants

NLP-based methods consume a sequence of tokens. Abstractly, each token contains two aspects of information, i.e., its token content and its position in the sequence. Take the most famous Transformer-based [38] models as an example. For each input token, its content as well as its position in the sequence are first transformed into two embedding vectors respectively and then summed, as shown in Figure 2a. And the resulting embedding set of tokens is fed to the subsequent networks. Thus, modifying either token content or token position results in different inputs of NLP-based models. If such modification does not change code semantics, these models need to learn from it.

Figure 2 shows three trivial semantically equivalent transformations that the sequence representation imposes on NLP-based models to learn. **Trans.1**, the first two instructions of this code snippet can be swapped without modifying the code semantics. However, this transformation results in changes in the produced embedding sets. For instance, the token embedding of the token `LOAD` is now added with the position embedding of the index `5` (as shown in Figure 2b) while it is originally added with the position embedding of the index `2`. Thus, the model needs to learn that *the order of some instructions can be adjusted without modifying the semantics while others cannot*. **Trans.2**, the entire code snippet is placed at a different position in the sequence, e.g., due to the insertion of some dummy instructions at the beginning of the sequence. In figure 2c, the position embeddings of all tokens are changed, resulting in a totally different embedding set. The model needs to learn that *the same sub-sequence of tokens in different positions are semantically equal*. **Trans.3**, all the uses of the register `r0` are substituted by a previously unused register `r2`. The model needs to learn that *the choice of exact registers used are semantics-independent*. On the contrary, the data flow passing through registers is an integral aspect of code semantics.

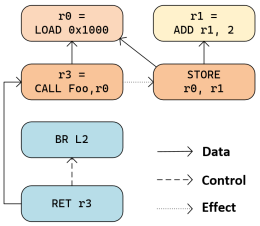
For these transformation rules to be learned by models and extended to real-world cases, a lot of related samples need to be fed. For instance, if only code snippets in Figure 2a and Figure 2c are given, models may learn that the given code snippet at position 0 and position k are semantically equivalent, while it is not necessary for them to learn if similar rules hold when another code snippet is given or the snippet is placed at another position. In addition, it costs extra network parameters and layers. Overall, learning these additional semantics makes the NLP-based approaches more expensive and more difficult to generalize.

2.3 Implicit Structures of Binary Code

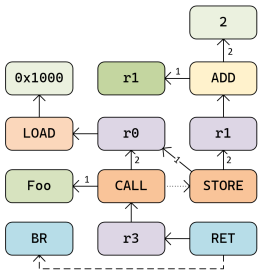
The transformation 3, mentioned in Section 2.2, suggests that the def-use relations between instructions constitute the entities of the code semantics. Apart from these relations,

```
1 L1:
2   r0 = LOAD 0x1000
3   r1 = ADD r1, 2
4   STORE r0, r1
5   r3 = CALL Foo, r0
6   BR L2
7 L2:
8   RET r3
```

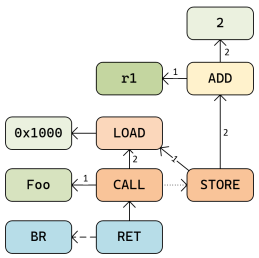
(a) Linear Representation in the toy IR



(b) ISCG: Instruction-based Semantics-Complete Graph



(c) TSCG: Token-based Semantics-Complete Graph



(d) SOG: Semantics-Oriented Graph

Figure 3: A proof-of-concept example of the step-by-step lifting of code from the linear representation to the Semantics-Oriented Graph.

binary code contains other semantic structures, which can be grouped into three categories: intra-instruction structures, inter-instruction relations, and function-level conventions.

Intra-instruction Structures. Instructions have internal structures. For instance, the instruction `add r1, r2, r3` in MIPS can be interpreted as the `add` operator uses the value cached in the register `r2` and `r3` and stores the produced value into the register `r1`. Meanwhile, the instruction `add rax, rdx` in x86-64 can be interpreted as the `add` operator uses the value cached in the register `rax` and `rdx`, but stores its outputs in both `rax` and `eflags` registers.

Inter-instruction Relations. Inspired by the ideas developed in the sea of nodes IR [37], we divide relations between instructions into three categories: data (def-use relations), control (branches), and effect (execution order).

Data relations reveal the fact that some instructions use values defined by others. Control relations define the control flow on the basic block level. Effect relations, which have been overlooked by previous work, establish restrictions on the execution order between instructions. Effect relations are necessary since data and control relations do not comprise the full code semantics. For instance, in the code snippet depicted in Figure 3a, no (explicit) data or control relations exist between the instructions on line 4 and line 5. If not introducing additional restrictions, we could swap these two instructions. However, this is dangerous, as the invocation of `Foo` may use or modify the same memory slot as the `STORE`.

To express the third type of relations, an alternative method that mimics the traditional control flow graph representation sequences the instructions in each basic block to restrict the execution order. However, this representation imposes excessive restrictions. For example, in Figure 3a, the instruction on line 4 cannot be swapped with any of the preceding two due to the constraint of the def-use relations (the instruction on line 4 uses `r0` and `r1`, which are modified by the instructions on line 2 and line 3 respectively). Meanwhile, the instructions on line 2 and line 3 can be swapped without affecting the code semantics, which suggests a more flexible representation.

For the pursuit of both effectiveness and efficiency, our goal is to incorporate only the *necessary* execution order into our final representation. Thus, we adopt the *effect* model, which models the additional execution order restriction as potential data flows. In the example given above, the `STORE` instruction modifies a memory slot, while the subroutine `Foo` invoked by the `CALL` instruction may read from or write to the same memory slot, which composes a potential data flow. To reveal such relations, inspired by the Click's IR [5, 6], we use an abstract temporary to represent each set of memory slots concerned. Instructions that may read values from or write values to that set of memory slots are considered to use or define the corresponding temporary.

It is worth noting that the constructed effect flows are related to the analysis ability. For instance, in the last example, if we figure out that the invoked subroutine does not interact with the memory or that the `STORE` instruction only modifies the current stack frame which will not be accessed by any subroutines, no effect relations needed to be introduced between them.

Function Level Conventions. Most real-world binary functions adhere to various conventions, including calling conventions, the layout of stack frames, and other details needed for a binary function to work properly on a certain system. These conventions are also indispensable parts of code semantics. Specifically, calling conventions define which registers are served as the call arguments and the return values, which aid in the recovery of the def-use relations of call instructions. Besides, functions store temporary variables in their own stack frames which will not be accessed by other functions. This understanding contributes to refining the effect relations.

2.4 Step to the Semantics-Oriented Graph

The semantics-preserving transformations listed in section 2.2 provide hints on properties that an ideal binary code representation should possess. First, the representation should not assign a position identifier to each instruction or token, as the semantics is independent of position. Therefore, a *graph representation* would be more appropriate. Second, the representation should not fully adopt the execution order to restrict instructions since a significant portion of execution order re-

strictions is machine-dependent but semantics-independent. Thus, we adopt the *effect model* to restrict only the necessary execution order. Third, some token contents are independent of semantics, while the relations between instructions or tokens carry the semantics. We thus aim to *eliminate the semantics-independent tokens* and *model semantic relations*.

Based on these analyses, we propose a graph-based binary code representation, named semantics-oriented graph (SOG). SOG reveals well-defined semantic structures, purges semantics-independent elements, and is capable of encoding implicit conventions of binary functions. Intuitively, SOG can be constructed from a linear representation in three steps:

First, we lift the sequence of instructions into a graph representation and reveal the relations between instructions as edges. Figure 3b shows an example of such a representation lifted from the code snippet in Figure 3a. This graph can be obtained by enhancing the data flow graph (DFG) with additional control flow and effect flow edges. We name this representation the instruction-based semantics-complete Graph (ISCG), as it carries exactly the same semantic information as the original linear representation and takes instructions as nodes. One interesting thing is that we do not mark which basic block an instruction belongs to, because instructions can actually float over basic blocks without changing the semantics, as long as these three relations are respected.

Second, we reveal intra-instruction structures by splitting instructions into tokens. This step simplifies the generation of node embeddings and eases the elimination of semantics-independent elements. The intra-instruction structures can be interpreted as another kind of data relations. For example, we can interpret the instruction `r0 = LOAD 0x1000` as the `LOAD` token using the `0x1000` token and the `r0` token using the processed result of the `LOAD` token. By further refining the inter-instruction relations on the exact instruction tokens which introduce them (i.e., control and effect relations are defined on instruction operators, while data relations are defined on the register tokens or others that temporarily cache data), and labeling the positions of the operands on the edges, we obtain the graph shown in Figure 3c. Edges with omitted labels in this graph have the default label `1` (i.e., these relations are built on their first operands). Figure 3c reveals semantically related inter-instruction relations and intra-instruction structures. We name this representation the token-based semantics-complete graph (TSCG).

Third, since the choice of temporary stores (e.g., registers or stack slots) to cache data between instructions is semantically independent, we further remove these nodes and connect their inputs and outputs directly, forming Figure 3d, the final representation that this study targets. It is worth noting that we preserve the use of uninitialized stores. Uninitialized stores mostly carry the value passed from the caller routine. If the calling convention of the current function is known, the names of uninitialized stores can be used to infer the position of the argument among the argument list. Thus, the uninitialized stores actually carry semantic information and are preserved.

SOG is capable of encoding various function-level conventions. For example, identified calling arguments can be revealed by simply adding data edges from the calling node to the argument nodes. To reveal the conventions of stack frames, we can model the effect using two abstract temporaries. One stands for the stack frame of current invocation and the other stands for all other memory regions. Thus, no effect relations will be built between the instructions that access only the current stack frame and the instructions that do not access it.

3 Details of SOG and its Construction [1]

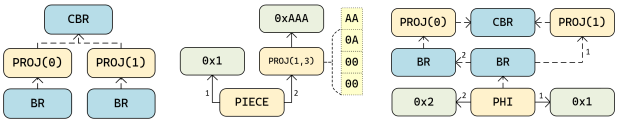
The construction of the SOG can be finished in a similar procedure that is used to convert linear IRs to the SSA form. To promote reader understanding, we divide the SOG into three subgraphs and detail their construction separately.

To construct the *control subgraph*, dummy `BR` instructions are inserted at the end of basic blocks when necessary to ensure that each basic block ends with a branch instruction. Branch instructions are then treated as prolocutors for basic blocks, and are lifted to the nodes of the graph. Control flow edges are added from branch targets to branches. To avoid ambiguity, each node is treated as defining only one value for each type of relation. For each conditional or indirect branch instruction that has multiple successors, we treat its outputs as a concatenated value and incorporate a `PROJ` node for each successor to project the concatenated value into a unit one for that particular successor. Figure 4a shows an example of such a case, where `CBR` denotes a conditional branch.

The *data subgraph* can be constructed in the procedure of def-use analysis. During the analysis of the def-use relations of each instruction, we first construct a node with the instruction operator as the node type, and then add directed edges, one per operand, from this node to some preceding node that defines the value of that operand. If such a preceding node cannot be found, that is, the operand is an integer literal or a register that has not been defined before, a new node that represents the integer literal or the uninitialized register is introduced. After that, we mark the corresponding operand as defined. In this way, all instructions referring to the same undefined operand connect to the same operand node.

Analyzing the def-use relations at the binary code level faces the alignment challenge. That is, the value in one operand may be defined by multiple instructions simultaneously. For instance, consider the following code snippet: `mov eax, 0xAAA; mov al, 1; mov edx, eax`. The value of the `eax` register referred by the third instruction is defined by both the preceding two instructions (in that code site, `eax=0xA01`). To address this issue, another type of node, named `PIECE`, is introduced to abstractly concatenate multi-

[1] Inspired by the sea of nodes IR [37] (the PROJ node), the P-code of Ghidra [31] (the PIECE node), and the Click's IR [6] (structure of PHIs).



(a) CBR+PROJ (b) PIECE (c) PHI

Figure 4: Examples of special nodes in SOG.

ple values. The lifted graph of the above code snippet can be found in Figure 4b.

The *effect subgraph* can be constructed in the same way as the data subgraph since the effect can be considered as a special kind of value that is used and defined by special instructions. Generally, two effect models can be identified [5,6]: the memory effect and the I/O effect. Specifically, instructions that may load value from memory are considered as using the memory effect. And instructions that may alter the state of the memory are considered as both using and defining the memory effect. According to this definition, `STORE` and `CALL` are the only instructions in the toy IR that both use and define the memory effect and the `LOAD` instructions also use the memory effect. The I/O effect can be constructed in a similar manner.

Phi Nodes. It is worth noting that the SOG is naturally of the static single assignment (SSA) form, as it defines one node for each instruction. To keep the conciseness, we need to introduce necessary `phi` instructions for both data values and effect values. Since the semantics of the `phi` instructions depends on the control flow, the first use of the `phi` node is set as the branch node of that basic block where the `phi` instruction resides in. Other uses of the `phi` node are set in the same order as the order of outgoing edges of the branch node. Thus, when a branch node receives control flow from its `i`-th outgoing edge, the `phi` node which uses that branch node can select the `i+1`-th input as the output. An example of `phi` node is shown in Figure 4c. The corresponding code snippet is `r0=0x1; CBR A, L3; L2: r0=0x2; BR L3; L3: XXX`.

Refer to Appendix A for the pseudo-code of our graph construction algorithm.

4 BCSD With SOG

4.1 Framework

Figure 5 illustrates the overall framework of our system. Each input binary function is first lifted to the proposed semantics-oriented graph, in which each node is attributed with a token and each edge has a type label and a position label. After normalization, we transform each node and edge into learnable embeddings. Following the common practice of previous work [16, 21, 44], we use GGNN [22] to aggregate neighbor structures of each node. Next, we employ the multi-head softmax aggregator to generate the graph embedding. Finally, we combine our graph embedding model into a Siamese network [3], enabling it to approximate the similarity score of a pair of functions by the similarity of their graph embeddings.

Training We adopt margin-based pairwise loss [21] with the distance-weighted negative sampling strategy [41] for training. Mini-batches are gathered by first sampling `N` different function symbols and then sampling `2` functions with different compilation settings for each symbol. The negative sampling process of each function g_a^b is formulated below, where a or k indicates the index of a function symbol in the mini-batch, b or l indicates the index of a binary function of that symbol, $\omega_{l,k}^{b,a}$ denotes the probability of choosing the sample g_k^l as the negative sample for g_a^b, and dim denotes the dimension:

$$\text{MiniBatch} : \{g_1^1, g_1^2, g_2^1, g_2^2, ..., g_N^1, g_N^2\} \tag{1}$$

$$w_{l,k}^{b,a} = -(dim-2)\mathrm{d}(g_k^l, g_a^b) - \frac{n-3}{2}(1-\frac{1}{4}\mathrm{d}(g_k^l, g_a^b)^2) \tag{2}$$

$$[\omega_{l,k}^{b,a}]_{l\in\{1,2\},\ k\neq a} = \mathrm{softmax}([w_{l,k}^{b,a}]_{l\in\{1,2\},\ k\neq a}) \tag{3}$$

We use the negative of cosine similarity as the distance metric (d in Eq. 2). The training loss is calculated as follows:

$$s_{i,j}^{pos} = 1 - m - \mathrm{sim}(g_i^j, g_i^{3-j}) \tag{4}$$

$$s_{i,j}^{neg} = \mathrm{sim}(g_i^j,\ \mathrm{neg\_sampling}(g_i^j)) - m \tag{5}$$

$$\mathcal{L} = \sum_{i=1}^{N}\sum_{j=1}^{2}(\ \max(s_{i,j}^{pos}, 0) + \max(s_{i,j}^{neg}, 0)\ ) \tag{6}$$

where $m \in [0,1]$ is a margin parameter, $j \in \{1,2\}$.

4.2 Graph Normalization and Encoding

Before we can harness the power of the graph neural network, we first need to transform node and edge attributes into embedding vectors. In SOG, each node is attributed with a token. We can directly map each token to a learnable embedding vector. Each edge has a type attribute (data, control, or effect) and a position attribute (the index of the corresponding operand). We separately transform the type and the position attributes into two learnable embeddings and add them to form the final edge embedding. The model needs to learn a vocabulary of tokens and labels.

However, too many different tokens (e.g. different integer literals) and position labels exist in binary code so it is nearly impossible to include all of them in the vocabulary and to require them to be presented in the training dataset. Thus, some tokens or labels are actually out of the vocabulary.

To address this problem, we assign each token a token type. Specifically, we identify three types of tokens: instruction tokens, integer literals, and register tokens. Since different architectures have different conventions in register use, we further divide register tokens into several sub-types according to the architectures. For each type of token, we identify the most common tokens and include them in the vocabulary

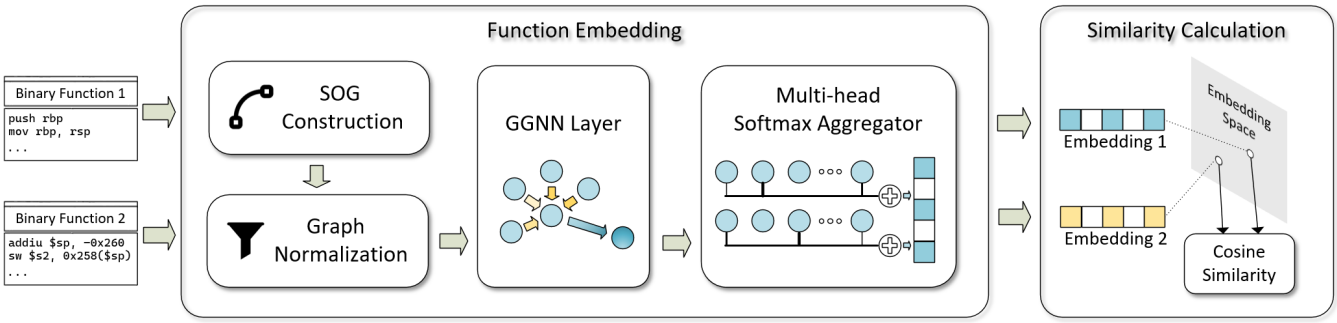


Figure 5: The overall framework of HermesSim.

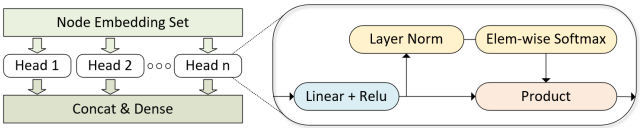


Figure 6: Structure of the proposed multi-head aggregation module.

of the model, and normalize other tokens. For instance, all obscure instruction tokens are now normalized as a general 'instruction token' and all rarely used integer literals are normalized as an 'integer literal'. In this way, the model can learn unified semantics for the less frequent tokens of each type.

4.3 Local Structure Capture

After normalizing and then encoding the SOG into an attributed graph, we use bidirectional GGNN layers [22] to capture the neighbor structures of each node. Let $\mathbf{h}_u^l$ denote the output embedding of node u at the l-th GGNN layer and $\mathbf{e}_{v,u}$ denote the edge embedding of the direct edge (v,u), then

$$\mathbf{m\_in}_u^l = \sum_{v \in In(u)} \mathrm{MLP}_{in}(\, \mathbf{h}_u^l \mid \mathbf{h}_v^l \mid \mathbf{e}_{v,u} \,) \tag{7}$$

$$\mathbf{m\_out}_u^l = \sum_{v \in Out(u)} \mathrm{MLP}_{out}(\, \mathbf{h}_v^l \mid \mathbf{h}_u^l \mid \mathbf{e}_{u,v} \,) \tag{8}$$

$$\mathbf{h}_u^{l+1} = \mathrm{GRU}(\, \mathbf{h}_u^l,\ \mathbf{m\_in}_u^l + \mathbf{m\_out}_u^l \,) \tag{9}$$

where $In(u)$ and $Out(u)$ denote sets of incoming and outgoing neighbors of node u respectively, the $\mid$ symbol denotes vector concatenation, MLP denotes the multilayer perceptron, and GRU denotes the gated recurrent unit.

4.4 Multi-head Softmax Aggregator

The output of r GGNN layers are a set of node embeddings $\{\mathbf{h}_u^r\}$ with each of them encoding r-hops neighbor structures. The final step of the function embedding module is to aggregate all node embeddings into a graph embedding. We adopt the proposed multi-head softmax aggregator, which is adapted from the softmax aggregator [19].

The softmax aggregator can be formulated as:

$$\mathrm{SoftmaxAggr}(\mathcal{H}|\mathbf{t}) = \sum_{\mathbf{h}_u^r \in \mathcal{H}} \frac{\exp(\mathbf{t} \odot \mathbf{h}_u^r)}{\sum_{\mathbf{h}_v^r \in \mathcal{H}} \exp(\mathbf{t} \odot \mathbf{h}_v^r)} \odot \mathbf{h}_u^r \tag{10}$$

where the vector $\mathbf{t}$ is the learnable parameters that control the softness and $\mathcal{H}$ is the set of all node embeddings in one graph. The final graph embedding is an elementwise weighted sum of node embeddings.

We extend the softmax aggregator by introducing multiple heads. Each head is allowed to learn a graph embedding in a different representation space and give attention to different sets of nodes and features. As shown in Figure 6, the input node embeddings of each head are first transformed into a different representation space through a Linear layer. A Relu layer is then applied to filter features deterministically, which encourages different heads to learn from a different set of features. Next, the Layer Normalization is applied to force each node to select some outstanding feature words while weakening some others, further encouraging node features to fuse in a more non-linear fashion. After that, the original Softmax aggregator is applied. Finally, the output embeddings of all heads are concatenated and another Linear layer is applied to generate the final embedding. The computation flow is formulated below:

$$\mathbf{x}_u^k = \mathrm{Relu}(\mathrm{Linear}_k(\mathbf{h}_u^r)) \tag{11}$$

$$\mathbf{g}^k = \mathrm{SoftmaxAggr}(\left\{ \frac{\mathbf{x}_u^k - \mathrm{E}[\mathbf{x}_u^k]}{\sqrt{\mathrm{Var}[\mathbf{x}_u^k] + \varepsilon}} | u \in \mathcal{V} \right\} | \mathbf{t}^k) \tag{12}$$

$$\mathbf{g} = \mathrm{Linear}(g^1 | g^2 | \ldots | g^h) \tag{13}$$

where $\mathcal{V}$ is the set of all nodes in the graph, k marks the index of the head, h is the number of total heads, and ε is a small value added for numerical stability. The computation of multiple heads can be processed in parallel.

5 Evaluation

In this section, we address the following research questions:

- **RQ1:** How effective is HermesSim in BCSD tasks compared to baselines? (§5.2)
- **RQ2:** Can SOG surpass the existing representations and the straw-man representations proposed in Section 2? (§5.3)
- **RQ3:** Can the multi-head softmax aggregator outperform other baseline aggregators? (§5.3)
- **RQ4:** How efficient is HermesSim in graph construction, training, inferring, and searching? (§5.4)
- **RQ5:** Can HermesSim effectively recall 1-day vulnerabilities in real-world scenarios? (§5.5)

We also investigate the effect of function sizes on the performances of HermesSim and other baselines (Appendix C).

5.1 Implementation and Experiment Setup

Our graph construction module is built upon the Ghidra [31], a powerful open-source decompiler. We utilize Ghidra to disassemble binary functions and lift them to its Pcode IR, based on which we then construct the SOG. Our graph embedding module is implemented using Pytorch [32] and PyG [13].

Normalization Parameters. For the instruction tokens, we select the 55 most frequent tokens from the total of 59 tokens appearing in the training dataset. As for the integer literals and register tokens, we include only those tokens that are present in at least one percent of functions in the training dataset.

Model Hyper-params. The number of GGNN layers is 6 and the message passing networks MLP_{in} and MLP_{out} are both of 3 layers. The node embedding size is 32 and the edge embedding size is 8. The graph embedding size of each aggregation head is 64 and the number of heads is 6.

Training Setup. For the training process, we use the method described in Section 4.1 to randomly sample each mini-batch. The batch size (N) is 80, and the margin (m) is 0.1. The `Adam` optimizer is used with a learning rate of 0.001. The initial random seed used to sample mini-batches remains the same across all training campaigns.

Dataset. We evaluate our method on Dataset-1 released by the previous study [27]. This dataset consists of 257K, 13K, and 522K binary functions in its training, validation, and testing set respectively. These functions come from three different architectures (x86, ARM, and MIPS), two different bitness modes (32 and 64 bits), 5 optimization levels (O0, O1, O2, O3, and Os), and are compiled by 2 different compiler families (GCC and CLANG), each with four different versions.

This dataset relies on IDA Pro 7.3 [35] to recover the control flow and the boundaries of basic blocks and then uses Capstone [34] to disassemble the raw bytes of each basic block. To minimize the potential impact of underlying disassembly capabilities, we use the control flow and the boundaries of the basic blocks presented in the original dataset, relying only on Ghidra to disassemble the raw bytes of each basic block.

To enable comparison with the previous work supporting limited architectures, we filter out a sub-dataset that contains only binary functions of the x64 architecture. These approaches are both trained and tested on the x64 sub-dataset. Others that support multiple architectures are trained on the whole dataset and also tested on the x64 sub-dataset.

Testing. We evaluate our method on the binary function retrieval task, i.e. given a query function f and a pool of N different binary functions, our goal is to retrieve the only ground truth function that is compiled from the same source code as the query function. We evaluate four subtasks: cross-architecture and bitness retrieval (XA), cross-optimization level retrieval (XO), cross-compiler, compiler version, and optimization level retrieval (XC), and cross-architecture, bitness, compiler, compiler version, and optimization level retrieval (XM). Additionally, we evaluate these subtasks with various pool sizes ranging from 2 to 10000. For each setting, we randomly select 1000 query functions.

Metrics. RECALL@1 and mean reciprocal rank (MRR) are used as our evaluation metrics. RECALL@1 is the percentage of queries that successfully recall the ground truth function as the most similar function. MRR is computed by:

$$MRR = \frac{1}{||Q||} \sum_{q \in Q} \frac{1}{r_q} \tag{14}$$

where Q is the set of all queries and r_q is the rank of the ground truth function of the query q.

5.2 Comparative Experiments

Baselines. We select the following baselines[2]:

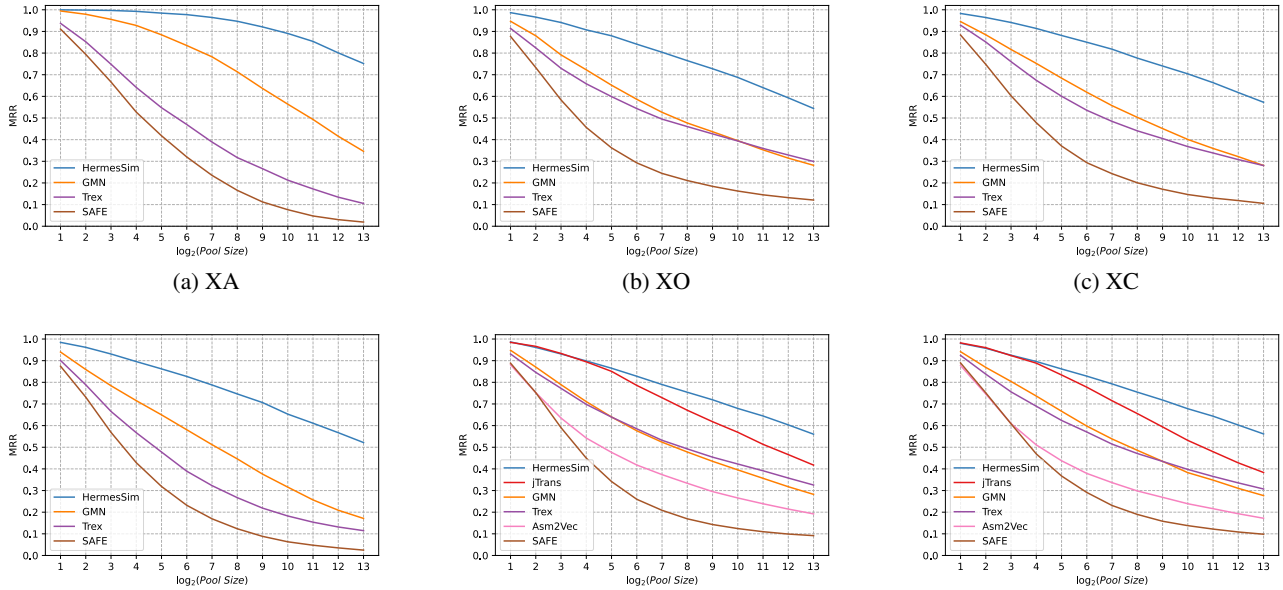
- *Asm2Vec* [9] proposes to model the CFG as multiple execution traces and then uses a variant of the PV-DM [30] NLP model to generate function embeddings. This method does not support cross-architecture tasks.
- *SAFE* [29] first builds an instruction2vec (i2v) model to map instructions to embedding vectors and then uses the self-attentive network proposed in [23] to aggregate instruction embeddings into a function embedding.
- *Graph Matching Network (GMN)* [21] is based on a variant of the GNN network that jointly reasons on a pair of CFGs. The existing study [27] shows GMN surpasses other GNN models and is among one of the state-of-the-art methods.
- *Trex* [33] utilizes the hierarchical Transformer based model to learn execution semantics from micro traces and transfer the learned semantic knowledge for BCSD tasks.
- *jTrans* [40], one of the state-of-art BSCD methods, manages to embed control-flow information into Transformer and pre-trains their model on a large-scale dataset consisting of about 21 million binary functions for training. This baseline supports only the x86 architecture.

For baselines other than jTrans, we use their original implementations with patches and settings provided in the previous

[2]Methods like Vulhawk [26] are not compared because their main contributions do not lie in enhancing the semantics learning.

Table 1: Results of the comparative experiments on the full dataset for XA, XO, XC, and XM subtasks, and on the x64 dataset for XO and XC subtasks (denoted as x64-XO and x64-XC respectively). XM and x64-XC subtasks are evaluated with two different pool sizes (i.e., 100 and 10000). Other subtasks are evaluated with a pool size of 100. The scores (%) are RECALL@1/MRR.

	XA	XO	XC	XM		x64-XO	x64-XC		N Params
		100		100	10000	100	100	10000	
SAFE	13.4/26.4	21.1/27.5	20.1/27.6	9.9/18.9	1.4/2.32	18.4/26.2	17.2/24.9	8.1/9.5	8.93M
Asm2Vec	-	24.6/30.1	25.8/31.7	-	-	31.8/37.7	29.0/35.0	13.5/16.6	-
Trex	31.2/42.1	46.8/53.1	45.4/52.5	24.4/34.4	8.6/11.1	51.5/57.7	45.9/53.2	26.2/30.1	61.8M
GMN	72.6/81.7	50.3/58.1	52.3/59.8	44.7/53.7	10.5/15.9	52.4/60.2	48.0/56.2	21.9/26.7	60.5K
jTrans	-	-	-	-	-	66.9/76.0	65.0/73.8	31.4/37.4	87.9M
HermesSim	**95.5/97.5**	**81.0/85.3**	**78.0/83.2**	**74.5/80.2**	**43.8/50.8**	**81.9/86.0**	**75.6/80.7**	**48.1/54.6**	388K



(a) XA (b) XO (c) XC

(d) XM (e) x64-XO (f) x64-XC

Figure 7: MRR scores on different pool sizes.

study [27]. To compare with jTrans, we finetune the released pre-trained model on our dataset using the default settings.

Results. As shown in Table 1, HermesSim outperforms all baselines by a large margin in all settings. Specifically, on the x64 dataset, HermesSim achieves a recall rate that is 22%, 16%, and 53% higher than the state-of-the-art approach, jTrans, in XO, XC (poolsize=100), and XC (poolsize=10000) experiments, respectively. As illustrated in Figure 7e and 7f, jTrans can only achieve comparative results to HermesSim when the pool size is extremely small (i.e., 16), which is not common in real-world applications. On the full dataset, HermesSim improves the recall by 132%, 161%, 149%, 167%, and 417% over GMN in XA, XO, XC, XM (poolsize=100), and XM (poolsize=10000) experiments, respectively.

NLP-based methods (SAFE and Trex) perform poorly in the XA subtask. GMN generally outperforms SAFE, Asm2Vec, and Trex, especially in the XA subtask. This can be attributed to the cross-architecture nature of CFGs. However, GMN performs worse than Trex in XO and XC subtasks when applying large pool sizes. One reason could be that CFGs only explicitly encode the control flow aspect, and are thus likely to collide when searching in large pools.

The last column of Table 1 shows the number of total parameters of each approach (excluding parameters of optimizers). NLP-based methods, namely, jTrans, Trex, and SAFE have at least one order of magnitude more parameters than graph representation-based methods. Trex and jTrans, which are based on large language models, have 159 and 226 times more parameters than our model, respectively.

Figure 7 demonstrates that HermesSim's performance remains more stable than baseline approaches as the pool size

Table 2: Results of the ablation study on the full dataset for XA, XO, XC, and XM subtasks, and on the x64 dataset for XO and XC subtasks. The second row lists the pool sizes. The scores (%) are RECALL@1/MRR.

		XA	XO	XC	XM		x64-XO	x64-XC	
			100		100	10000	100	100	10000
MSoft	CFG-OPC200	92.8/95.6	67.7/73.9	67.1/73.2	62.9/69.3	32.0/38.7	69.9/75.7	64.2/70.3	36.8/42.9
	CFG-PalmTree	-	-	-	-	-	70.8/76.4	66.1/72.3	36.3/43.0
	CFG-HBMP	94.3/96.6	69.5/75.8	68.9/74.9	65.3/71.9	33.7/40.6	72.0/77.4	67.2/73.0	39.0/45.5
MSoft	P-DFG	93.4/96.2	76.4/81.6	74.1/79.8	69.7/76.0	37.4/44.7	76.3/81.6	69.9/75.7	42.8/49.3
	P-CDFG	94.6/96.9	77.2/82.5	75.3/80.8	71.1/77.3	38.6/45.9	77.2/82.4	72.4/77.8	43.6/50.5
	P-ISCG	95.1/97.2	78.0/82.9	74.8/80.3	71.3/77.2	40.1/47.2	78.1/82.6	71.9/77.1	44.2/50.9
	P-TSCG	95.5/97.4	79.2/83.6	76.2/81.7	73.2/78.9	41.3/48.8	79.1/83.7	73.5/78.7	46.4/52.9
Set2Set		89.5/93.0	67.2/72.2	64.5/69.9	61.2/66.9	29.8/36.6	69.2/73.7	60.7/66.1	37.3/43.1
Softmax	P-SOG	90.9/94.3	72.9/78.1	71.0/76.6	64.3/71.2	32.6/39.4	74.0/78.7	66.9/72.6	41.2/47.2
Gated		93.5/96.1	75.6/80.3	72.5/77.6	68.3/74.2	38.4/44.9	76.3/80.7	69.0/74.3	43.9/49.9
MSoft	P-SOG	**95.5/97.5**	**81.0/85.3**	**78.0/83.2**	**74.5/80.2**	**43.8/50.8**	**81.9/86.0**	**75.6/80.7**	**48.1/54.6**

increases from 2 to 8192. The relative performance of HermesSim becomes even better than the state-of-the-art baselines (i.e., jTrans and GMN) as the pool size increases (e.g., from 4.5 to 35.0 in XM subtask compared to the GMN as the pool size increases from 2 to 8192).

5.3 Ablation Study

Baselines. To demonstrate the efficiency of the SOG representation for BCSD, we select several promising representations from previous work that are not surpassed by others:

- *CFG-opc200* is a CFG based binary function representation using the `opc200` manually crafted features as basic block attributes. This representation is proposed by Marcelli et al. [27] and shows advantages over previous Word2vec [30] based methods [28].
- *CFG-PalmTree* aggregates unsupervised instruction embeddings generated by the PalmTree model [20] as the basic block embedding. PalmTree is the state-of-the-art unsupervised instruction embedding network. We follow the practice of the original paper to use mean pooling as the aggregator to obtain basic block embedding. This baseline only supports the X86 architecture.
- *CFG-HBMP* use HBMP model [36] to compute the block embedding end-to-end. This method is proposed by Yu et al. [44] and performs better than previous pre-training methods [28, 43] using Bert [7] or Word2vec [30].

In addition, we compare the SOG representation with straw-man representations mentioned in Section 2.4:

- *P-DFG*[3] takes instructions as nodes and def-use relations between instructions as edges. We build the DFG based on Ghidra's Pcode IR and generate instruction embeddings through a bidirectional GRU layer end-to-end.
- *P-CDFG* is similar to the P-DFG representation except that it integrates the control flow relations as additional edges.
- *P-ISCG* is the first semantics-complete graph proposed in Section 2.4. The difference between ISCG and CDFG lies in the introduction of effect flow edges. An example of ISCG can be found in Figure 2b.
- *P-TSCG* is the second semantics-complete graph proposed. Compared to the ISCG, it additionally reveals intra-instruction structures. Figure 2c is an example of TSCG.

Our multi-head softmax aggregator is compared with:

- *Softmax Aggregator* [19] is the base model of our proposed aggregator. It is a generalized version of both the Mean Aggregator and the Max Aggregator.
- *Gated Aggregator* is proposed along with the GGNN by Li et al. [22] and is used by several previous studies [21, 27].
- *Set2set Aggregator* is proposed by Vinyals et al. [39] and is used by the related work [44]. This aggregator is based on the iterative attention mechanism.

To compare with those baselines, we keep other parts of HermesSim, tune only the hyper-parameters tightly related to these methods, and report the best results found. For representations, we tune the hyper-parameters of graph encoders and the batch sizes within the constraint of GPU memory. For aggregators, we tune the hyper-parameters inside these modules, the final graph embedding size, and the batch sizes.

Observing that the result scores of some baselines are close, we repeat the experiments in this subsection 10 times to mitigate the effect of randomness. Mean values are reported.

Results. Table 2 shows the results of the ablation study. The first section of the table contains the results of baseline representations, in which the CFG-HBMP method stably outperforms the other two methods. The second section shows the results of the straw-man representations mentioned in Sec-

[3] The 'P-' prefix refers to 'Pcode-based'.

tion 2.4. The third section tests the effectiveness of baseline aggregators on the proposed SOG representation. The last section shows the results of our proposed method.

Compared to the baseline representation in the first section of Table 2, the proposed SOG representation improves the recall by around 10 percent in all subtasks except XA. XA is the easiest subtask for the GNN and the graph representation-based methods, in which even the CFG-opc200 method can achieve a recall rate as high as 92.8%. And we observe that the relative performance of the SOG over the CFG-HBMP becomes better as the pool size increases from 100 to 10000 in both the XM subtask (from 9.2% to 10.1% in RECALL@1) and the x64-XC subtask (from 8.4% to 9.1% in RECALL@1).

In the second section, the RECALL@1 and MRR scores of P-DFG, P-CDFG, P-ISCG, and P-TSCG generally increase successively, demonstrating that the reveal of control flow relations, effect flow relations, and intra-instruction structures can indeed improve the efficacy. The introduction of effect flow edges slightly hurt the performance in the XC subtask on both datasets when setting the pool size as 100. This may be attributed to the dirty effect problem which we will discuss later in Section 6. The performance in all subtasks benefits from the introduction of control flow edges and the reveal of the intra-instruction structures.

The P-DFG method outperforms the CFG-HBMP method in nearly all subtasks except XA, which demonstrates the superiority of CFGs in the cross architectures scenario and the superiority of DFGs in the cross-optimization and the cross-compiler subtasks. In addition, the superiority of the SOG over the TSCG supports the hypothesis that keeping the semantics-independent elements in the representation hurts the generalization ability of the model.

The multi-head softmax aggregator achieves the RECALL@1 and MRR scores by 134% and 129% respectively over the original softmax aggregator on the XM subtask with a pool size of 10000. Additionally, the multi-head softmax aggregator significantly outperforms other baseline aggregators.

Figure 8 illustrates how the MRR scores of different representations in the validation dataset increase during the training campaigns. The SOG representation achieves the best MRR score throughout the training campaigns. The ISCG achieves better scores than TSCG in the early stages but fails to retain its advantage later. The CFG-HBMP method, which utilizes the HBMP NLP model to learn basic block attributes, achieves comparable validation scores as the P-CDFG method but fails to generalize as well to the larger testing dataset.

5.4 Runtime Efficiency

Table 3 shows the runtime cost of HermesSim. The lifting time is the time cost to lift a binary function from the linear representation to the SOG. The inferring time consists of encoding the SOG to a numeric embedding vector. The searching time is the cost of comparing a function embedding against a pool of other embeddings.

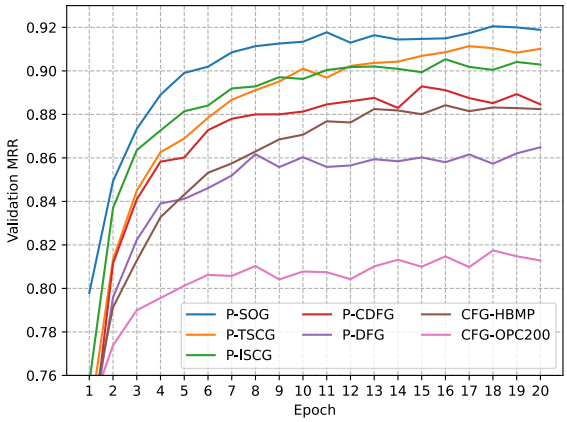


Figure 8: Validation MRR scores of different representations on the XM task.

Table 3: Efficiency of HermesSim in terms of average time cost per 10^3 functions in our testing dataset. Lifting and searching time is evaluated on Intel(R) Xeon(R) Silver 4214 @2.20GHz CPU but in a single process. Training and inferring time are evaluated on one NVIDIA RTX3080 GPU.

Training	Lifting	Inferring	Searching
62mins (20 epochs)	3.20s	0.35s	1.50ms

Table 4: Average number of nodes and edges in different graph representations and the inferring time cost.

	Num. Nodes	Num. Edges	Inferring
CFG(-HBMP)	30.6	40.8	0.14s
P-DFG	427.7	442.7	0.20s
P-CDFG		551.7	0.25s
P-ISCG		688.9	0.27s
P-TSCG	803.2	1310.7	0.54s
P-SOG	542.7	1010.7	0.35s

In addition, we show the sizes of different graph representations as well as their inferring time cost in Table 4. CFGs and ISCGs are smaller than SOGs on average, which implies both lower memory and time costs during the local structure capture stage, but at the cost of complicating the node attributes extraction. SOGs have 32% fewer nodes and 23% fewer edges than TSCGs and are 35% faster in terms of the inferring time, showing the purge of semantics-independent elements meaningfully improves the efficiency as well.

5.5 Real-world Vulnerability Search

In this experiment, we collect 12 RTOS firmware images from three vendors (TP-Link, Mercury, and Fast) and perform the 1-day vulnerability search task. We first build a repository that consists of all functions in these images and manually identify 5 CVEs and 10 related vulnerable functions in the TP-Link WDR7620 firmware. Then we use these vulnerable functions as queries to search for similar functions in the repository. Our repository contains 62605 functions in total, which are of two different architectures, i.e., ARM32 and MIPS32.

For each query function, we categorize the functions in the repository into three groups: **c1:** functions built from exactly the same source code as the query function. These functions are previously unknown vulnerable functions that need to be retrieved. **c2:** functions built from a slightly different source code (e.g., from different versions) but are of the same symbol as the query function (when the function symbol is available). Functions in this categorization are potentially vulnerable functions that need manual identification. **c3:** other functions (i.e., functions that are compiled from different source code). We identify the ground truth by performing similarity searches using HermesSim and other compared baselines and manually examine the top 20 results of all methods.

We evaluate the results using RECALL@1 and MRR as well. Our goal is to retrieve all functions in **c1** and **c2**. Ideally, the BCSD system should rank functions in **c1** before functions in **c2** and **c3**, and rank functions in **c2** before those in **c3**. Thus, we calculate the rank of each ground truth function by adding the number of more dissimilar functions that have higher similarity scores by 1. For example, for a ground truth function in **c1**, its rank is the number of functions that have higher similarity scores but are categorized in **c2** and **c3** plus one. The ideal rank of each ground truth function is 1.

Table 5 shows the results of HermesSim and other baselines that support the cross-architecture task. HermesSim outperforms other baselines in terms of the number of failures, RECALL@1, and MRR by a large margin. HermesSim can handle most cases except that it ranks a slightly modified version of the function 7 before three functions in **c1**.

In the SA scenario, SAFE, Trex, and GMN perform poorer on recalling functions in **c2**. For instance, GMN can recall 29 out of 31 functions in **c1**, but can only recall 6 out of 11 functions that have slight modifications at the source code level. This indicates that baseline methods are ineffective in ranking functions according to the semantics similarity. In contrast, HermesSim can recall all of them.

Furthermore, NLP methods, i.e., SAFE and Trex, are unable to recall any of the 43 ground truth functions that are of an architecture different from the query function (i.e., ineffective in the XA scenarios). Even GMN can only recall 2 of these functions due to the large pool size. In contrast, our proposed method can recall on average 38.3 such functions.

See Appendix B for details about the vulnerabilities.

Table 5: Results of real-world vulnerability searching experiments. The Tot. columns show the number of ground truth functions in two categories respectively for each query function. The SAFE, Trex, GMN, and Our columns represent the number of ground truth functions that *fail to be recalled* by each respective method.

#	Tot.		SAFE		Trex		GMN		Our[a]	
	c1	c2	c1	c2	c1	c2	c1	c2	c1	c2
0	7	1	3	1	3	0	3	0	0.0	0.0
1	1	8	0	8	0	4	0	5	0.0	0.0
2	9	0	5	0	4	0	4	0	0.0	0.0
3	3	2	2	2	1	2	1	2	0.0	0.0
4	4	7	2	7	1	5	2	7	0.5	0.1
5	4	7	2	7	1	7	2	7	0.4	0.5
6	6	4	2	4	2	4	1	4	0.2	0.0
7	7	5	4	4	3	4	3	3	3.0	0.0
8	2	3	0	3	0	3	0	3	0.0	0.1
9	3	2	1	2	1	2	1	0	0.0	0.0
XA[b]	15	28	15	28	15	28	15	26	4.1	0.7
SA[b]	31	11	6	10	1	3	2	5	0.0	0.0
Tot.	46	39	21	38	16	31	17	31	4.1	0.7
R1[c]			54	3	65	21	63	21	91	98
MRR			55	3	67	22	73	21	94	99

[a] As with previous experiments in Section 5.3, the results of our method is derived from averaging the results of 10 independent runs.
[b] XA: Cross-architecture. SA: Single Architecture.
[c] RECALL@1 (%)

6 Discussion

The semantics-oriented graph is a concise and semantics-complete graph representation for binary code. Although we only focus on utilizing this representation for BCSD in this paper, it should also be applied to other binary code-related tasks that require understanding the code semantics. Besides, this representation is very suitable for encoding multiple extra analysis abilities. We discuss potential improvements to this representation in more detail below.

Dirty Effect Problem. The effect flow in the source code is mostly kept as is during compilation and optimization because the compiler does not know what will happen if the execution order of the effect-related instructions is changed. Thus, due to its cross-optimization and cross-architecture nature, such effect flow should be useful for BCSD.

However, some effect-related instructions are introduced during compilation, such as the use of the stack frame, which is transparent to the compiler and can be manipulated. Meanwhile, the number of stack temporary variables is architecture and optimization level dependent. Thus, including stack memory accesses in the effect flow may not be beneficial. An additional pass of the load-store elimination analysis can be

applied to clean up this pollution, which is left as future work.

I/O Effect Model. We only investigate the memory effect model in this paper. The I/O effect model pertains to instructions that interact with input/output (I/O) devices, such as the LOAD instructions that read from the memory-mapped I/O regions. Different from the memory effect model, receiving data from I/O devices can change the states of these devices as well. The I/O effect model should be useful when the tested functions directly communicate with I/O devices.

Extra Information and Encoding Ability. References to strings, integers, external function symbols, and other related entities are not handled by this study. They can be integrated by properly developing modules that encode these features into a unified embedding space as other node tokens. NLP-based techniques should be suitable for encoding these elements, as demonstrated in previous studies [18, 44].

Addressing Analysis Failures. Currently, traditional program analysis algorithms may fail to recover the full control flow relations of the indirect branches, which is still an open problem. This limitation affects not only our SOG representation but also the linear representation used by NLP-based methods and the CFG representation. In one scenario, due to control flow recovery failures, basic blocks at the potential indirect branch targets are not included in the scope of the function. This causes all representations to be equally incomplete. In the other scenario, only the control flow relations are lost while the affected basic blocks are detected (e.g. because other paths lead to them). The linear representation is insensible to such failures because it does not exploit the control flow relations, while the CFG and the SOG primarily miss some edges. Nevertheless, this does not imply that the linear representation is superior because it offers no more information than graph representations to neural networks for inferring the missed control flow.

We argue that it is impossible to understand the code semantics without first figuring out the control flow. And therefore, it may be worthwhile to explore how to provide the necessary information (e.g., referred data) to deep neural networks to enable them to deduce the control flow of indirect branches.

7 Related Work

This section surveys the learning-based BCSD approaches. We divide the development in this direction into three lines.

Deep learning (DL) for BCSD. With the development of artificial intelligence techniques, DL algorithms with more powerful feature extraction capabilities are applied to BCSD. For example, GNN is widely used due to its ability to effectively capture structural information [15, 21, 24, 42, 44]. Xu et al. [42] introduce the `Structure2vec` model to learn features from control flow structures and achieve significantly better results than previous methods. Li et al. [21] propose the use of more advanced GNN models, i.e., the GGNN and the GMN.

Recently, some researchers leverage more advanced NLP techniques and pre-trained models to automatically extract latent representations of binary code at either the basic block level or function level. Zuo et al. [45] model the basic blocks as natural sentences and design a cross-assemble-lingual basic block embedding model based on word embedding and LSTM. Perdisci et al. [29] combine the skip-gram method [30] and the self-attentive network [23] to generate function embeddings. Yu et al. [43], Guo et al. [16] and Luo et al. [26] specially design several pre-training tasks for binary code and train a BERT-based [8] large language model to generate basic block embeddings. Pei et al. [33], Ahn et al. [1], and Wang et al. [40] use a Transformer-based model to generate function-level embeddings directly. However, these methods treat the disassembled binary code as natural languages and fail to exploit the well-defined code semantics. Besides, the use of increasingly large models results in significantly higher training and inference costs.

Binary Code Representation for BCSD. The raw representation of a binary function is a stream of bytes, which is featureless. Thus, it is crucial to design an effective representation for BCSD. In the early stage, CFG is widely used for binary code representation due to its cross-architecture nature [2, 10, 14]. Later, Gao et al. [15] enhance the CFG with inter-basic-block data flow edges. Guo et al. [16] manage to learn an embedding from DFG along with CFG and make use of the def-use relations of code. Wang et al. [40] embed control flow information into Transformer through parameter sharing and get promising results. Previous work has taken an involuntary step toward exploiting full code structures. Our work examines full code semantics and reveals it through a novel graph representation. Similar representations [5, 6, 12, 37] exist in compiler research to ease the optimization, but they are built from the source code and thus cannot be directly employed in binary code related tasks.

BCSD beyond the Code. Other studies attempt to integrate related information beyond the code semantics. For instance, Yu et al. [44] embed string and integer references separately and concatenate them into the final function embedding. Kim et al. [18] introduce the binary disassembly graph to include external function references and string literal references as features. Luo et al. [26] first predict the compiler and the optimization level using an entropy-based technique, and then transfer function embeddings from different compilation settings into a unified embedding space. While our work focuses on exploiting the code semantics, techniques developed in this line can be integrated into our system as well (see §6).

8 Conclusion

In this paper, we propose a semantics-complete binary code representation, the semantics-oriented graph (SOG), for BCSD. This representation not only exploits the well-defined

code structures, semantics, and conventions but also purges the semantics-independent elements embedded in the low-level machine code. We detail the construction of the SOG and discuss potential improvements to this representation.

To unleash the potential of the SOG for BCSD, we propose a novel multi-head softmax aggregator, which allows for the effective fusion of multiple aspects of the graph. By integrating the proposed techniques, we build an effective and efficient BCSD solution, HermesSim, which relies on the GNN model to capture structural information of the SOG and adopts advanced training strategies.

Extensive experiments demonstrate that HermesSim significantly outperforms state-of-the-art approaches in both laboratory experiments and real-world vulnerability searches. In addition, our evaluation proves the value of revealing full semantic structures of binary code and cleaning up semantics-independent elements. We also demonstrate the effectiveness of the proposed aggregator and the efficiency of HermesSim.

Acknowledgments

We thank the anonymous reviewers of this work for their helpful feedback. We thank Marcelli et al. [27] for their valuable work in reviewing and reproducing the previous work. This research was supported, in part, by National Natural Science Foundation of China under Grant No. 62372297, Ant Group Research Fund, Science and Technology Commission of Shanghai Municipality Research Program under Grant No. 20511102002, National Radio and Television Administration Laboratory Program (TXX20220001ZSB002). Yuede Ji was supported by the University of North Texas faculty startup funding.

References

[1] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 361–374, Austin TX USA, December 2022. ACM.

[2] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. SIGMA: A Semantic Integrated Graph Matching Approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, March 2015.

[3] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature Verification using a "Siamese" Time Delay Neural Network. In *Advances in Neural Information Processing Systems*, volume 6. Morgan-Kaufmann, 1993.

[4] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing*, 11(4):307–317, 2014.

[5] Cliff Click. From Quads to Graphs: An Intermediate Representation's Journey. *Technical Report CRPC-TR93366-S, Center for Resesearch on Parallel Computation, Rice University*, 1993.

[6] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *ACM Sigplan Notices*, 30(3):35–49, 1995.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[9] Steven H. H. Ding, Benjamin C. M. 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, San Francisco, CA, USA, May 2019. IEEE.

[10] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. *Sstic*, 5(1):3, 2005.

[11] 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, Vienna Austria, October 2016. ACM.

[12] 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*, 9(3):319–349, July 1987.

[13] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[14] Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In Liqun Chen, Mark D. Ryan, and Guilin Wang, editors, *Information and Communications Security*, volume 5308, pages 238–255. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[15] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 896–899, Montpellier France, September 2018. ACM.

[16] Yixin Guo, Pengcheng Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Exploring GNN based program embedding technologies for binary related tasks. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 366–377, Virtual Event, May 2022. ACM.

[17] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 63–72, New York, NY, USA, 2011. Association for Computing Machinery.

[18] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 151–163, Virtual South Korea, July 2022. ACM.

[19] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. Deepergcn: All you need to train deeper gcns. *arXiv preprint arXiv:2006.07739*, 2020.

[20] 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*, pages 3236–3251, Virtual Event Republic of Korea, November 2021. ACM.

[21] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *Proceedings of the 36th International Conference on Machine Learning*, pages 3835–3845. PMLR, May 2019. ISSN: 2640-3498.

[22] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[23] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A STRUCTURED SELF-ATTENTIVE SENTENCE EMBEDDING. In *International Conference on Learning Representations*, 2017.

[24] Shangqing Liu. A unified framework to learn program semantics with graph neural networks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1364–1366, Virtual Event Australia, December 2020. ACM.

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

[26] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *Proceedings 2023 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2023. Internet Society.

[27] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, Boston, MA, August 2022. USENIX Association.

[28] Luca Massarelli, Giuseppe A. Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In *Proceedings 2019 Workshop on Binary Analysis Research*, San Diego, CA, 2019. Internet Society.

[29] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 11543, pages 309–329. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.

[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[31] NSA. Ghidra. https://ghidra-sre.org/.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[33] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Learning Approximate Execution Semantics From Traces for Binary Function Similarity. *IEEE Transactions on Software Engineering*, 49(4):2776–2790, April 2023. Conference Name: IEEE Transactions on Software Engineering.

[34] Nguyen Anh Quynh. Capstone - the ultimate disassembly framework. https://www.capstone-engine.org/.

[35] Hex Rays. Ida pro. https://hex-rays.com/ida-pro/.

[36] Aarne Talman, Anssi Yli-Jyrä, and Jörg Tiedemann. Sentence Embeddings in NLI with Iterative Refinement Encoders. *Natural Language Engineering*, 25(4):467–482, July 2019.

[37] Google V8 Team. Turbofan. https://v8.dev/docs/turbofan.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[39] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[40] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jTrans: jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, Virtual South Korea, July 2022. ACM.

[41] Chao-Yuan Wu, R. Manmatha, Alexander J. Smola, and Philipp Krahenbuhl. Sampling Matters in Deep Embedding Learning. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2859–2867, Venice, October 2017. IEEE.

[42] 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, Dallas Texas USA, October 2017. ACM.

[43] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1145–1152, April 2020.

[44] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. *NeurIPS*, page 12, 2020.

[45] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society.

A Algorithm for the Construction of SOG

Listing 1 shows the overview structure of our graph construction algorithm, where the $defState$ is an elaborate interface that provides all the control, data, and effect definition states at the current code point. The $defState$ supports putting variables with partial overlap, in which case it divides the variables into smaller units that either do not overlap or are exactly the same. And it is able to piece multiple defined variables to form a larger requested variable. For each defined variable, the $defState$ maintains a stack to record recent definitions on that variable. Meanwhile, it logs all definition operations to support commit and revert. The MayWrapWithProj procedure wraps the use of conditional or indirect branches that have multiple successors, as discussed in Section 3.

B Details about the Vulnerability Searching

Table 6 details the vulnerable functions found and examined in the real-world vulnerability search experiment (§5.5). The results show that even models from different vendors share a significant amount of closed-source code bases. These findings highlight the pressing need for reliable binary code similarity detection techniques.

Furthermore, we find the function symbols in the TL-WDR7620 firmware. While other firmware images such as TL-SG2206 and FAST-FAC1200RQ do not contain these symbols. With the help of the BCSD techniques, we can transfer the function symbols found in one firmware image to others, making reverse engineering much easier.

Algorithm 1 SOG Construction Algorithm

```
1: procedure ConstructSOG(f)
2:     ▷ f: A binary function in assembly languages or linear IRs.
3:     defState ← new DefState();
4:     g ← new SOGConstructor();
5:     effectNodes ← GetAbstractEffectNodes();
6:     dt ← GetDominatorTree(f.cfg);
7:     f ← InsertPhiNodes(f, dt);
8:     f ← PrepareControl(f);
9:     ProcessBlock(g, defState, f.start, effectNodes);
10:    return g;
11:
12: procedure PrepareControl(f, defState)
13:    for bb in f do
14:        if not bb ends with a BRANCH instruction then
15:            bb.append(new DummyBranch());
16:        node ← SOGNodeFromInst(bb.getLastInst());
17:        defState.put(bb, node);
18:
19: procedure ProcessBlock(g, defState, bb, effectNodes)
20:    defState.commit();        ▷ Commit current definition state.
21:    BuildInBlock(g, defState, bb, effectNodes);
22:    for child in dt.getChildren(bb) do
23:        ProcessBlock(child);
24:    defState.revert();        ▷ Revert to the last commit point.
25:    return ;
26:
27: procedure BuildInBlock(g, defState, bb, effectNodes)
28:    for inst in bb do
29:        node ← SOGNodeFromInst(inst);
30:        g.add(node);
31:        ▷ Recover data flow relations.
32:        for input in inst.inputs do
33:            inpNode ← defState.peekOrNew(input);
34:            node.addDataUse(inpNode);
35:        for output in inst.outputs do
36:            defState.put(output, node);
37:        ▷ Recover effect flow relations.
38:        for effect in effectNodes do
39:            if UseEffect(node, effect) then
40:                node.addEffectUse(defState.peek(effect));
41:            if DefineEffect(node, effect) then
42:                defState.put(effect, node);
43:        ▷ Handle control flow relations.
44:        if IsControlNode(node) then
45:            for pred in bb.predecessors do
46:                prenode ← defState.peek(pred);
47:                prenode ← MayWrapWithProj(prenode, bb);
48:                node.addControlUse(prenode);
49:    for succ in bb.successors do
50:        inOrder ← succ.getPrecedingIndex(bb);
51:        for phi in succ.phiNodes do
52:            phi.setPhiUse(inOrder, state.peekOrNew(phi.store));
53:    return ;
```

```
// Original Vulnerabile Function.
int sub_404D34C4(int a1, int a2,
                 unsigned __int16 a3)
{
  if ( !a1 || !a2 || !a3 )
    return 1;
  ......
}
// The slightly different version.
int sub_40477EB0(int a1, int a2,
                 unsigned __int16 a3,
                 int a4, int a5)
{
  if ( !a1 || !a2 || !a3 || !a4 || !a5 )
    return 1;
  ......
}
```

Figure 9: False Positive Samples.

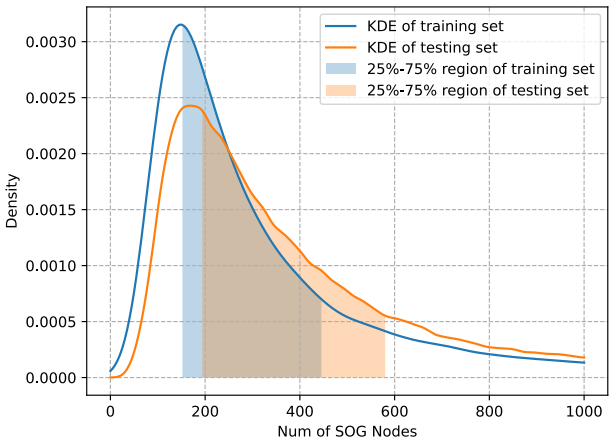


Figure 10: Kernel density estimation of the number of SOG nodes in the training and testing sets respectively.

False Positive Case Figure 9 shows the only example that consistently appears as a false positive in 10 independent runs of HermesSim. The slightly modified version function is ranked above the function of the same source code but of a different architecture. All compared baselines, i.e., SAFE, TREX, and GMN fail as well in this case. This problem should be mitigated by additionally introducing slightly modified function pairs in the training dataset (e.g., by mutating the source code).

Table 6: Details of 1-day vulnerability search.

#	CVE ID	Function symbol	Confirmed #	Affected firmware
0	CVE-2022-26987	MmtAtePrase	8	TL-AP1908GI, TL-WDR7620, MERCURY-D196G, TL-AP1902GP, TL-WDR5650, FAST-FAC1200RQ, MERCURY-M6G, TL-AP1903GP[a]
1	CVE-2022-26988	tWlanTask	9	TL-WDR7620, TL-AP1908GI[a], TL-AP1903GP[a], TL-AP1207GI[a], TL-AP1902GP[a], TL-WDR5650[a], MERCURY-M6G[a], MERCURY-D196G[a], FAST-FAC1200RQ[a]
2	CVE-2022-26988	MmtAte	9	TL-WDR7620, TL-AP1902GP, MERCURY-D196G, TL-AP1908GI, TL-AP1903GP, TL-WDR5650, TL-AP1207GI, MERCURY-M6G, FAST-FAC1200RQ, [a]
3	CVE-2021-33373	uninstallPluginReqHandle	5	TL-WDR7620, MERCURY-D196G, TL-WR886N, TL-WDR5650[a], FAST-FAC1200RQ[a]
4	CVE-2020-28877	protocol_handler	11	TL-WDR7620, MERCURY-D196G, TL-AP1908GI, TL-WR886N, FAST-FAC1200RQ[a], TL-WDR5650[a], TL-SG2206R[a], TL-WA933RE[a], TL-AP1903GP[a], TL-AP1902GP[a], TL-AP1207GI[a]
5	CVE-2020-28877	ms_idle_handler	11	TL-WDR7620, MERCURY-D196G, TL-WR886N, TL-AP1908GI, TL-SG2206R[a], TL-WDR5650[a], FAST-FAC1200RQ[a], TL-WA933RE[a], TL-AP1902GP[a], TL-AP1903GP[a], TL-AP1207GI[a]
6	CVE-2020-28877	parse_advertisement_frame	10	TL-WDR7620, MERCURY-D196G, TL-AP1908GI, TL-AP1902GP, TL-AP1903GP, TL-WR886N, FAST-FAC1200RQ[a], TL-SG2206R[a], TL-AP1207GI[a], TL-WDR5650[a]
7	CVE-2020-28877	parse_discovery_frame	12	TL-WDR7620, MERCURY-D196G, TL-AP1903GP, TL-AP1908GI, TL-SG2206R, TL-AP1207GI, TL-WR886N, TL-AP1903GP[a], TL-AP1902GP[a], TL-AP1207GI[a], FAST-FAC1200RQ[a], TL-WDR5650[a]
8	CVE-2021-33374	phCenterXmlHandle	5	TL-WDR7620, MERCURY-D196G, TL-WDR5650[a], FAST-FAC1200RQ[a], TL-WA933RE[a]
9	CVE-2021-33374	isHttpDataValid	5	TL-WDR7620, MERCURY-D196G, TL-WA933RE, TL-WDR5650[a], FAST-FAC1200RQ[a]

[a] Identified functions in these firmware images are in category c2, i.e., they are from slightly different versions of source code.

C Impact of Function Sizes

Distribution of function sizes. Figure 10 exhibits the distribution of function sizes in the training and testing sets in terms of the number of SOG nodes. The peaks of the two KDEs are both around the 25% thresholds, with values of 148 and 168 for the training and testing sets, respectively. Since the functions of the training and testing sets come from different projects, these curves should resemble the real-world distribution of functions.

Performance on sets with extremely large and extremely small functions. Another two sub-datasets are created for testing: x64-XC-Small, and x64-XC-Large, which include only the top 1% small functions (have less than 71 nodes) and only the top 1% large functions (have more than 3676 nodes) in the testing set. Due to the limited number of functions in these sub-datasets, we sample only 200 functions as queries for each task and use a pool size of 100.

As demonstrated in Table 7, a significant performance improvement can be observed for all approaches when limiting the queries to be extremely small or large functions. This seems intuitive since distinguishing those outlier functions from others is easy. When further applying this restriction to functions in pools, the performance suffers due to the increased similarity between negative functions and the queries.

SAFE, Trex, and GMN all demonstrate inadequacies in handling large functions. Asm2Vec shows improved performance in recalling large functions compared to small functions in general pools but struggles to do so in pools with similarly sized functions. jTrans exhibits the opposite behavior of Asm2Vec. HermesSim behaves similarly to jTrans, but with better scalability in handling large functions. We believe the improved ability of HermesSim to handle large functions (relative to GMN) should be owed to the abundant semantics of SOG and the powerful aggregator.

The insufficiency of NLP-based methods (e.g., Trex and jTrans) in handling large functions may stem from their truncation mechanism (they receive only fixed-length input and discard any portion of the sequence that exceeds this limit).

Table 7: Results of the study on the impact of function sizes. Queries and pools are separately sampled from three testing sub-datasets: x64-XC, x64-XC-Small, and x64-XC-Large. The scores (%) are MRR.

Quries sampled from / Pools sampled from	xXC [a] xXC	xXC-S [b] xXC	xXC-L xXC	xXC-S xXC-S	xXC-L xXC-L
SAFE	24.9	45.4	37.4	27.8	24.4
Asm2Vec	35.0	69.0	76.8	60.7	56.2
Trex	53.2	91.9	72.6	81.9	67.0
GMN	56.2	89.9	86.3	65.0	58.4
jTrans	73.8	93.4	87.8	79.9	82.4
HermesSim	80.7	97.8	96.2	85.2	92.6

[a] xXC stands for 'x64-XC' in this table, we omit x64 for conciseness.
[b] S and L are abbreviations for 'Small' and 'Large'.