VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search

Zhenhao Luo, Pengfei Wang[✉], Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu and Kai Lu
College of Computer, National University of Defense Technology
{zh.luo, pfwang, bswang, ytang, xiewei, zhouxu, liudanjun12, kailu}@nudt.edu.cn

***Abstract*—Code reuse is widespread in software development. It brings a heavy spread of vulnerabilities, threatening software security. Unfortunately, with the development and deployment of the Internet of Things (IoT), the harms of code reuse are magnified. Binary code search is a viable way to find these hidden vulnerabilities. Facing IoT firmware images compiled by different compilers with different optimization levels from different architectures, the existing methods are hard to fit these complex scenarios. In this paper, we propose a novel intermediate representation function model, which is an architecture-agnostic model for cross-architecture binary code search. It lifts binary code into microcode and preserves the main semantics of binary functions via complementing implicit operands and pruning redundant instructions. Then, we use natural language processing techniques and graph convolutional networks to generate function embeddings. We call the combination of a compiler, architecture, and optimization level as a *file environment*, and take a divide-and-conquer strategy to divide a similarity calculation problem of C_N^2 cross-file-environment scenarios into $N-1$ embedding transferring sub-problems. We propose an entropy-based adapter to transfer function embeddings from different file environments into the same file environment to alleviate the differences caused by various file environments. To precisely identify vulnerable functions, we propose a progressive search strategy to supplement function embeddings with fine-grained features to reduce false positives caused by patched functions. We implement a prototype named VulHawk and conduct experiments under seven different tasks to evaluate its performance and robustness. The experiments show VulHawk outperforms Asm2Vec, Asteria, BinDiff, GMN, PalmTree, SAFE, and Trex.**

I. Introduction

Code reuse is widespread in software development. However, masses of codes and libraries are reused into multiple architecture binaries without security audit, which leads to many vulnerabilities hidden in software projects. Synopsys audited 2,409 projects in 2021, reporting that 97% of projects contained third-party code, and 81% of them contained known vulnerabilities [49]. A single vulnerability in the open-source code may spread across thousands of software, exposing millions of people to serious software security threats.

Unfortunately, with the development and deployment of the Internet of Things (IoT), the harms of code reuse are magnified. IoT devices are widely used in various scenarios. For different usage requirements, these IoT firmware images from different instruction set architectures (ISAs) are generated by different compilers with different optimization levels. However, numerous IoT firmware images only provide binary files, which have no source code available for security analysis. Their symbol information, such as function names, is generally stripped. Thus, binary code search becomes an active research focus for seeking vulnerabilities hidden in IoT devices.

Binary code search is applied to find similar or homologous binary functions in a large function repository. It is widely used in vulnerability detection [5]–[7], [16], [29], [41], [43], [48], [58]. For example, given a binary file, the binary code search compares its functions with all functions in the vulnerability repository based on function similarity to seek vulnerable functions in the binary file. In addition, it is also used in malware analysis [2], [4], [13], [18], [20] and binary patch analysis [19], [22], [56]. Since IoT firmware images come from different compilers, optimization levels, and ISAs, this brings tough challenges to binary code search, which requires high robustness for search approaches.

P1: Seeking vulnerabilities in IoT firmware requires binary code search methods robust across ISAs. In the mono-architecture binary code search, Asm2Vec [10], DeepBinDiff [11], and PalmTree [27] using natural language processing (NLP) techniques have achieved encouraging results. However, they can only search binary code on the same ISA and do not support cross-architecture tasks. InnerEye [60] treats binaries from different ISAs as different natural languages and uses neural machine translation to calculate binary code similarity. SAFE [35] trains its language model using binaries from multiple ISAs to search binary code across architectures. These rely heavily on training data and are difficult to implement for multiple ISAs. Lifting architecture-specific binary code to an architecture-agnostic intermediate representation (IR) is an effective way to address the cross-architecture challenge in IoT firmware. However, natural language and IR have essential differences [42]. Unlike natural language, IR contains `EFLAGS` as implicit operands (e.g., ZF). These flags control the function's execution paths and have important implications for function semantics. In addition, many redundant instructions in IR reduce the main semantics' weights, impacting the extraction precision of the main semantics.

P2: IoT firmware images are compiled by different compilers with various optimization levels due to usage requirements. Binaries from the same source code compiled by different compilers with optimization levels have similar semantics but different structures, which brings great challenges to binary code search. In this paper, we consider 3 architectures (`x86`, `arm`, and `mips`), 2 word sizes (`32-bit` and `64-bit`),

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24415
www.ndss-symposium.org

2 compilers (Clang and GCC), and 6 optimization levels (O0, O1, O2, O3, Os, and Ofast), totaling 72 combinations ($3 \times 2 \times 2 \times 6$). If binaries are selected from any two above combinations, there are a total of 2,556 (C_{72}^{2}) combined scenarios. The existing methods [9], [10], [27], [41] pin their hopes on deep learning to alleviate these differences and build a robust model against these scenarios. It may be possible to build a robust model for one or several specific scenarios. However, building a robust model against these 2,556 scenarios is complicated. Also, no information directly indicates compilers and optimization levels in binary files.

To solve the aforementioned problems, in this paper, we propose a novel cross-architecture binary code search approach named VulHawk. It contains a novel intermediate representation function model (IRFM) to generate robust function embeddings. In the IRFM, we first lift binary code to microcode. Then, we treat microcode sequences as language and use a variant of the RoBERTa model [31] to build basic block embeddings. We adopt graph convolutional networks (GCNs) to integrate basic block embeddings and control flow graphs (CFGs) to generate function embeddings. For the cross-architecture challenge in **P1**, the microcode is an architecture-agnostic language, which allows our model to be trained from one ISA and to search functions in multiple ISAs. For redundant instructions and implicit operands in **P1**, we implement an instruction simplification in the IRFM. We consider the assignments to implicit operands (EFLAGS) as real assignment instructions, which can help IRFM complement the implicit operand semantics. For redundant instructions, the instruction simplification simplifies microcode based on def-use relations, which prunes redundant instructions and preserves the main semantics of binary functions. This helps IRFM extract function semantics more precisely. We also propose root operand prediction (ROP) and adjacent block prediction (ABP) pre-training tasks to help the model understand relations between operands and data-flow relations between basic blocks.

For the challenges in **P2**, we take a divide-and-conquer strategy to divide the similarity calculation problem with 2,556 scenarios into 71 embedding transferring issues. We refer to the combination of the compiler, architecture, and optimization level as a *file environment*. Faced with 72 file environments, we choose an intermediate file environment, and transfer function embeddings from different file environments into the same file environments to alleviate differences. First, we introduce Shannon's entropy [47] from an information-theoretic perspective to represent the amount of information in binary files. In our practice, we find that binary files from the same file environments have similar entropy distributions. So we use entropy streams to identify file environments. With knowledge about functions' file environment, we deploy an entropy-based adapter to transfer these function embeddings into the intermediate file environment to alleviate differences caused by different environments. Furthermore, we propose a progressive search strategy to search candidate functions for keeping retrieval high efficiency and precision. First, it uses function embeddings to retrieve top-K function candidates based on the Euclidean distance. Then, we propose a similarity calibration method, which supplements the function embeddings using fine-grained features to reduce false positives.

In summary, we have made the following contributions:

- We propose an IRFM to generate robust function embeddings across architectures. It lifts binary code into microcode and preserves the main semantics of binary functions via instruction simplification. Two pre-training tasks are proposed to help our model to learn the root semantics of operands and grasp block data-flow relations. We use GCNs to integrate basic block embeddings based on CFGs to generate function embeddings. The IRFM can work across different architectures, which resolves the problem **P1**.
- According to the divide-and-conquer strategy, we use entropy streams to identify the file environments of binary files from the information-theoretic perspective. We propose an entropy-based adapter to transfer function embeddings into the same file environment to alleviate differences caused by different file environments. This makes our model robust against different compilers and optimization levels, resolving the problem **P2**.
- We propose a progressive search strategy, which implements a similarity calibration using fine-grained features to boost the performance and reduce false positives caused by patched functions.
- We implement VulHawk and evaluate it in three different scenarios: one-to-one comparison, one-to-many search, and many-to-many matching, across compilers, optimization levels, and architectures. The experiments show that VulHawk outperforms the state-of-the-art methods.
- We release the program and the pre-trained model of VulHawk (https://github.com/RazorMegrez/VulHawk) to facilitate the follow-up research.

II. Background

A. Problem Definition

Cross-architecture binary code search aims to retrieve top-K semantically similar candidate functions for massive binary functions extracted from various IoT devices [55]. Inspired by existing work [10], [55], [57], we define that two binary functions are semantically similar if they are compiled from the same or logic-similar source code. Like binary code similarity detection, the core of binary code search is the design of a robust model to detect whether given functions are similar. Instead of one-to-one matching, binary code search considers one-to-many search, which requires methods to retrieve semantically similar candidates more quickly and accurately. In the real world, IoT firmware can be compiled by various compilers (e.g., GCC and Clang) with different optimization levels (e.g., O0-O3, Os, and Ofast), which leads to compiled binary functions with the same semantics but different structures. Therefore, an effective cross-architecture binary code search needs to achieve the following goals:

- Cross-architecture support. Since IoT devices may come from different architectures, it requires methods to be robust against the differences introduced by different architectures.
- Cross-compiler support. Due to different development environments, compilers and compiler versions are usually different. It requires methods to tolerate the syntactic variations introduced by different compiler back-ends.
- Cross-optimization support. In the real world, the same source code may be compiled with different optimization levels due to various requirements (e.g., faster or more

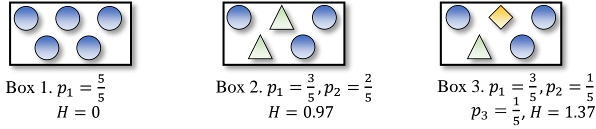


Fig. 1. Examples of Shannon's entropy calculation.

refined). It requires methods to be robust against the control flow structure change by different optimization levels.
- High precision and efficiency. Binary code search should accurately retrieve the most similar functions from millions of functions and distinguish irrelevant ones. For a large function database, it requires binary code search methods to search similar functions more quickly.

B. Entropy theory

In the information-theoretic perspective, Shannon's entropy [47] is a classic information measurement. It measures the randomness and the average amount of information in a system. The entropy function is as follows:

$$H = \int_{\mathcal{S}} -p(x) \log_2 p(x)\, dx \tag{1}$$

where $\mathcal{S}$ is the set of elements, and $p(x)$ represents the probability of element x. Fig. 1 illustrates examples of Shannon's entropy calculation. Each pattern represents a different element. For example, Box 1 is full of circles whose entropy H is 0; Box 2 has circles and triangles, which is more complex than Box 1, and its entropy is higher than Box 1; Box 3 is the most complex system of the three, whose entropy H is the highest. Through entropy analysis, we can get an up-front understanding of the average amount of information in a system before we drill down into it. In the binary code search task, we get the information distribution of binary files through the binary file entropy, which can infer information such as their compilers and optimization levels. This helps our model choose suitable parameters for different input binaries.

III. Design

In this section, we describe VulHawk's design. It contains three components: an intermediate representation function model, an entropy-based adapter, and a progressive search strategy. Fig. 2 shows the overview of VulHawk.

The IRFM is to generate basic block embeddings and function embeddings. We first lift binary code to microcode. Then, the instruction simplification complements the implicit operand semantics and prunes redundant instructions, which can preserve the main semantics of functions and improve the robustness of the IRFM. After that, we use a language model based on RoBERTa [31] to build basic block embeddings. During model training, we propose root operand prediction and adjacent block prediction pre-training tasks to let the IRFM understand relations between operands and data-flow relations between basic blocks. Finally, we adopt GCNs to aggregate neighbor basic block embeddings to capture control-flow relations to generate function embeddings.

The entropy-based adapter identifies file environments of input binary files and takes a divide-and-conquer strategy to make function embeddings from different file environments more similar. Here, we introduce entropy from an information-theoretic perspective. We first use entropy to predict the file environments, and then use the entropy-based adapter to transfer function embeddings into the intermediate file environment according to their file environments to mitigate the differences introduced by file environments.

The progressive search strategy is used to precisely detect candidates for function queries. We propose a two-step strategy, including coarse-grained search and similarity calibration. With the help of similarity calibration, we filter false positives (e.g., patched functions) when detecting vulnerable functions, which makes our model more precise.

TABLE I. Microcode category

Microcode	# of types	Type list
opcode	73	nop, stx, ldx, ldc, mov, neg, lnot, bnot, xds, xdu, low, high, add, sub, mul, udiv, sdiv, umod, or, and, xor, smod, shl, shr, sar, cfadd, ofadd, cfshl, cfshr, sets, seto, setp, setnz, setz, setae, setb, seta, setbe, setg, setge, setl, setle, jcnd, jnz, jz, jae, jb, ja, jbe, jg, jge, jl, jle, jtbl, ijmp, goto, call, icall, ret, push, pop, und, ext, f2i, f2u, i2f, u2f, f2f, fneg, fadd, fsub, fmul, fdiv.
operand	16	mop_z, mop_r, mop_n, mop_str, mop_d, mop_S, mop_v, mop_b, mop_f, mop_l, mop_a, mop_h, mop_c, mop_fn, mop_p, mop_sc

A. Intermediate Representation Function Model

To resolve the cross-architecture challenge, we lift binary code into IR and propose an Intermediate Representation Function Model (IRFM) using RoBERTa model [31] to build IR embeddings. The IRFM embeds IR functions into high-dimensional embedding space, which makes embeddings of semantically similar functions close in the numerical space.

1) Intermediate Representation: For binaries from various architectures, we disassemble them and lift binary code to an architecture-agnostic IR. We use IDA Pro [45] and its IR, named microcode, in our implementation, but other disassemblers and IRs also work (e.g., McSema [40]). As Table I shows, the microcode groups categorize various instructions from different architectures into 73 opcodes and 16 types of operands. For example, `mop_z` represents none operands, `mop_r` represents registers, and `mop_str` represents string constants[1]. Microcode, a well-established IR, can mitigate the impacts of instruction type differences on the cross-architecture binary code search.

Tokenization. In microcode, an instruction consists of one opcode and one operand triplet which includes left, right, and destination operands. Unlike PalmTree [27] which splits an instruction into fine-grained basic elements (e.g., "mov", "[", "+" and "qword"), we split an instruction into one opcode and three operands (i.e., `left`, `right` and `dest`) based on the characteristics of microcode. The base address and offsets are different in different binaries, which introduce noises and make the model less robust. We normalize these addresses (e.g, `0x4040E0` and `0x4150D0`) with a special token `[addr]`. To alleviate the OOV (Out-Of-Vocabulary) problem, we introduce 16 root-operand tokens according to

[1]More details can be found at https://www.hex-rays.com/products/decompiler/manual/sdk/hexrays_8hpp_source.shtml

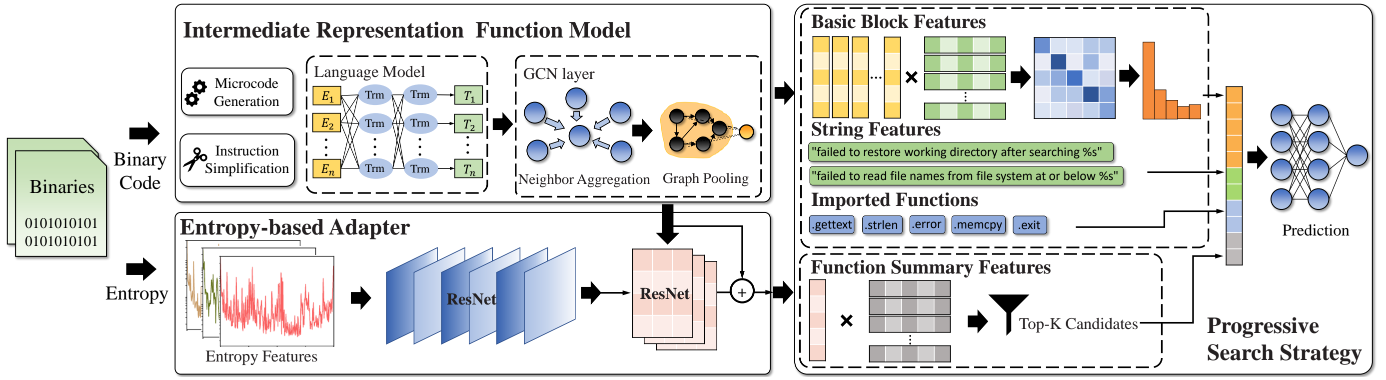


Fig. 2. The overall workflow of VulHawk.

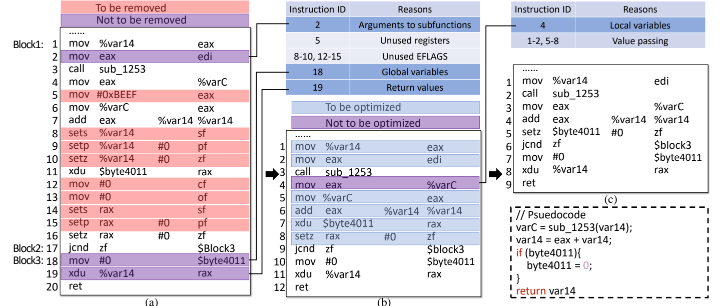


Fig. 3. Detailed steps of the instruction simplification.

microcode operand types in Table I. For operands in the vocabulary, we use their own tokens, and for those OOV operands, we use root-operand tokens to represent their basic semantics. In the pre-training phases, we replace tokens, whose frequency is less than 100 times, with their root-operand tokens to build root-operand token embeddings.

2) Language Model: The IRFM is based on RoBERTa [31], the state-of-the-art and widely used NLP model. The model uses a multi-layer bidirectional transformer encoder to build embeddings. Due to these essential differences between microcode and natural language, we make the following necessary improvements for the IR language model.

Token Type Layer. Unlike natural language, microcode consists of opcodes and operands, not just single words. Opcodes represent the operations to be performed (e.g., `ldx` and `goto`), and the operands indicate the data or memory location used for the operations. Considering these differences, we use a token type layer to help IRFM distinguish between opcodes and operands. We divide tokens into three types: `opcode`, `operand`, and `others`. The `others` type includes special tokens (e.g., `[pad]`) with no actual semantics.

Instruction Simplification. Binary code contains `EFLAGS` (i.e., flag registers) as implicit operands. These `EFLAGS` are implicitly assigned by instructions and used as inputs to conditional jumps. They control function execution paths and have important implications for function semantics. Unfortunately, most static BCSD approaches ignore these `EFLAGS` [10], [11], [41], [53], [59], [60]. Although PalmTree [27] mentions `EFLAGS`, it does not handle EFLAGS in terms of the instruction sequences. Dynamic binary analysis approaches, e.g., VEX [39], to emulate the real program behaviors, have to consider all `EFLAGS`. In static BCSD approaches, important semantics will be lost if EFLAGS are not considered; when all `EFLAGS` are considered, redundant `EFLAGS` not only introduce extra overhead, but may obscure the main semantics of binary code. Therefore, we only consider the used `EFLAGS`. We convert the assignment of each instruction to implicit operands into a real assignment instruction via microcode (e.g., Ln.12 in Fig. 3(a)), and preserve their used `EFLAGS` into instruction sequences.

Fig. 3(a) shows a microcode sequence, including global variables, return values, arguments to subfunctions, and redundant instructions. Redundant instructions (e.g., Ln.12-15 in Fig. 3(a)) bring extra overheads to handle and reduce the weights of the function's main semantics, impacting the embedding generation performance. When pruning redundant instructions, we prevent deleting global variables, arguments to subfunctions, and return values, because these instructions represent function behaviors and include essential semantics.

Here, we propose an instruction simplification based on def-use relations to prune redundant instructions and preserve important semantics. First, we mark the following "important" instructions to avoid deletion: (1) Global variables and local variables are stored in memory instead of registers, so we mark assignment instructions whose destination operand is a memory address, e.g., Ln.18 in Fig. 3(a). (2) Return values are usually stored in the specific registers, e.g., `rax` (x86) and `x0-x1` (arm). Thus, we mark specific registers based on calling conventions near the return instruction on all paths, e.g., Ln.19 in Fig. 3(a). (3) The arguments to subfunctions appear before the function call, and they are not overwritten by other instructions before being passed into the subfunction, e.g., Ln.2 in Fig. 3(a). We use loose rules to "important" instructions to ensure that no main semantics are mistakenly deleted. We consider instructions whose defined registers or `EFLAGS` are not used in subsequent instructions as `unused` instructions (e.g., Ln. 5, 12-15 in Fig. 3(a)) and prune them. After pruning `unused` instructions, we optimize redundant instructions (e.g., Ln. 5-8 in Fig. 3). We focus on the instructions that define a register directly assigned to another variable, named value passing instructions. Through instruction simplification, the 20 instructions of Fig. 3(a) are simplified to 9 instructions, which preserves the main semantics of Fig. 3(a) and is similar to its pseudocode. This helps IRFM extract more precise function semantics. In practice, the RoBERTa model accepts limited input length, and the instruction simplification enables the input of RoBERTa to preserve more valid instructions.

3) Pre-training Tasks: In the training phase, we use Masked Language Model (MLM), Root Operand Prediction (ROP), and Adjacent Block Prediction (ABP) for pre-training.

Masked Language Model. We introduce an MLM model to understand relationships between microcode and build suitable word embeddings. The MLM is first introduced by BERT [8], which uses the context tokens surrounding a mask token to predict what the masked token to optimize model parameters. In the mask layer of MLM, it randomly selects 15% of tokens to replace. For selected tokens, 80% of them are replaced by `[mask]` token, 10% are replaced with their root-operand tokens (the opcode root is itself), and 10% are unchanged. Fig. 4 shows an example, where yellow, red, and green boxes represent masked tokens, replaced tokens, and predicted results, respectively. In Fig. 4, opcode `setz` and register `r0` are masked as `[mask]`, and immediate number constant `#0` is replaced with its root-operand token `mop_n`.

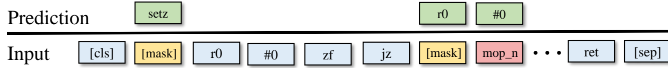


Fig. 4. Masked Language Model (MLM)

During training, we feed the final hidden states corresponding to the masked/replaced tokens into an output softmax over the vocabulary to predict the probability of these tokens. The loss function of MLM uses the Cross-Entropy loss as follows:

$$\mathcal{L}_{MLM}(\theta_1, \theta_2) = -\sum_{i \in M} \log p(y_i = \hat{y}_i | \theta_1, \theta_2), \hat{y}_i \in \{1, 2, ..., |V|\} \quad (2)$$

where y is the ground truth, $\hat{y}$ is the prediction, $|V|$ is the size of vocabulary, and M denotes the masked token set. θ_1 and θ_2 are the parameters of the IRFM and MLM head, respectively.

Root Operand Prediction. We propose the ROP pre-training task to associate token semantics with their root token semantics, which makes our model generate more reliable root token semantics for OOV words. In microcode, operands are divided into 16 types (see Table I). We use these as root-operand tokens. It is friendly to OOV operands because we can convert OOV operands into their root-operand tokens to represent their root semantics. For example, a specific address `0xdeadbeef`, assuming that is an OOV operand, our model assigns it with the semantics of root token `mop_a` which represents address operands, while the traditional model cannot distinguish its semantics [10], [35]. Since the opcode-root tokens are themselves, the ROP task will not predict the root opcode tokens. We perform an ROP head to predict their root tokens. In the training phase, we feed the final hidden states of tokens into a linear transformation. We use an output softmax over the operand types to predict the probability of these root-operand tokens. The loss function of ROP uses the Cross-Entropy loss as follows:

$$\mathcal{L}_{ROP}(\theta_1, \theta_3) = -\sum_{i \in S} \log p(y_i = \hat{y}_i | \theta_1, \theta_3), \hat{y}_i \in \{1, 2, ..., |V_R|\} \quad (3)$$

where $|V_R|$ is the size of root-operand vocabulary, and S denotes the operand token set. θ_1 and θ_3 are the parameters of the IRFM and ROP head, respectively.

Adjacent Block Prediction. In binary functions, there are data-flow relations between basic blocks. Unlike natural language, variables in binary code are required to be defined before being used. The basic blocks with data-flow relations are order-sensitive, which is not directly captured by the IRFM. To train a model that understands data-flow relations between adjacent blocks, we propose an ABP pre-training task. Specifically, given two basic blocks A and B, where B is a successor of A, variable x is defined in block A, and variable x is used in block B. We label the order of A-B as `positive` and the order of B-A as `negative`. Note that A and B are not the same blocks, and A is not the successor of B. Also, we do not consider these cases that A and B have only control-flow relations without data-flow relations. Because if there is no support of data-flow relations, the reverse order of blocks may also occur. We feed the final hidden state of token `[cls]` in the IRFM into the ABP head, a linear transformation, to identify whether the input two microcode sequences are in positive order. The loss function of ABP uses the Cross-Entropy loss as follows:

$$\mathcal{L}_{ABP}(\theta_1, \theta_4) = -\sum_{i \in \mathcal{D}} \log p(y_i = \hat{y}_i | \theta_1, \theta_4) \quad (4)$$

where y is the order label (`positive` or `negative`), $\hat{y}$ is the prediction, and $\mathcal{D}$ denotes the training set. θ_1 and θ_4 are the parameters of the IRFM and ABP head, respectively.

The total loss function of the language model is the combination of the above three loss functions:

$$\mathcal{L}_{LM} = \mathcal{L}_{MLM} + \mathcal{L}_{ROP} + \mathcal{L}_{ABP} \quad (5)$$

4) Function Embedding Generation: The task of IRFM in VulHawk is to generate function embeddings. First, we generate basic block embeddings. For input microcode blocks, the IRFM transformer encoders output sequences of hidden states. Here, we apply a mean pooling layer to integrate microcode instruction embeddings. According to the pre-trained model results [54], the last layer's hidden states are too close to the target tasks (e.g., MLM) during pre-training, which may be biased to these pre-training tasks. The hidden states of the second-last layer provide more generalization than those of the last layer. Therefore, we use mean pooling on the hidden states of the second-last layer to generate basic block embeddings.

The existing studies [34], [55] already showed that solutions based on CFGs have advantages in cross-architecture scenarios. Here, we integrate basic block embeddings and CFGs to generate function embeddings. Considering the multi-branch structure of binary functions, we use GCNs [24] to capture CFG structures and aggregate basic block semantics to their neighbor basic blocks. We consider binary functions as attributed graphs, where their basic blocks are nodes in the graphs, and their embeddings are attributes of nodes. We feed the attributed control flow graphs (ACFGs) into the GCN layer. $X^{(l)}$ represents features of the l-th layer nodes, and the aggregation function is as follows:

$$\begin{aligned} &\tilde{A} = A + I_N, \qquad \tilde{D}_{ii} = \sum \tilde{A}_{ij} \\ &X^{(l+1)} = ReLU(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X^{(l)} W^{(l)}) \end{aligned} \quad (6)$$

Here, $\tilde{A}$ is the adjacency matrix with self-connections. I_N

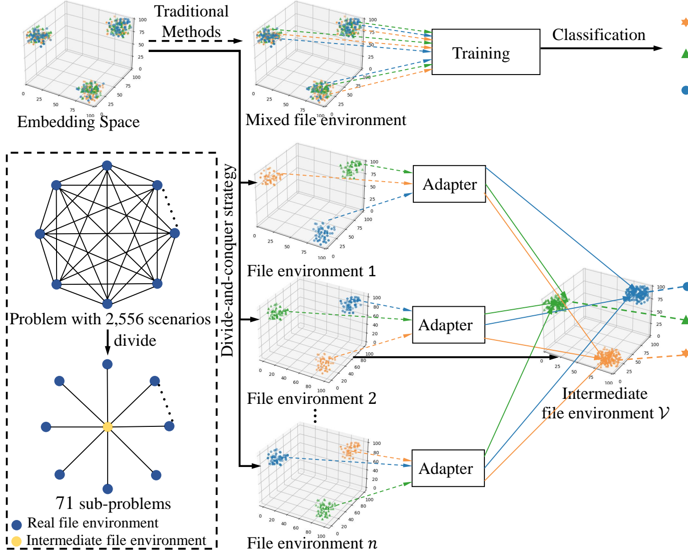


Fig. 5. The diagram of divide-and-conquer strategy.

is an identity matrix. $\tilde{D}_{ii}$ is a degree matrix of the node, and $W^{(l)}$ is a layer-specific trainable weight matrix. ReLU denotes an activation function. After an aggregation layer, the blocks learn contextual semantics from their adjacent blocks. To comprehensively learn block semantics and structures of CFGs, we adopt a 16-layer GCN to aggregate neighbor semantic embeddings. Finally, we use a mean pooling on the output of the GCN layer to generate function embeddings.

Training. Given two binary functions, we generate the ground truth y, i.e., dissimilar (0) and similar (1), based on function names and source files. We use Euclidean distance to calculate the similarity s of two functions as follows:

$$s = \frac{1}{1 + \text{Distance}(X_1, X_2)} \tag{7}$$

The training objective is to make the similarity of similar functions approach 1, and the similarity of dissimilar functions approach 0. We use the Cross-Entropy loss as the loss function:

$$\mathcal{L}_{Function} = -\sum_{i=0}^{N}\sum_{j=0}^{N}(y_{ij}\log(s_{ij}) + (1 - y_{ij})\log(1 - s_{ij})) \tag{8}$$

where y is the ground truth, s_{ij} is the similarity of functions i and j. We use the Adam optimizer to optimize the GCN's parameters to minimize the loss $\mathcal{L}_{Function}$.

B. Entropy-based Adapter

In the real world, binary functions are compiled by multiple compilers with different optimization levels from various architectures. In this paper, we call the combinations of compilers, architectures, and optimization levels as file environments. Functions from different file environments, even from the same source code, may differ in instructions and structures.

Divide-and-conquer. Fig. 5 shows an example of matching similar functions in the embedding space. Given an embedding space, points with the same color indicate similar functions and their variants. Existing methods [10], [14], [27], [55] do not distinguish the file environments of functions, and build one model to generate embeddings for binary functions in a mixed file environment. Embeddings of different functions have good discrimination in the same file environment, but mixed file environments may cause embedding collisions, which greatly increases the complexity of binary code similarity search. Furthermore, the differences between file environments are different. For example, the differences between O0 and O3 optimizations and the differences between GCC and Clang compilers are different. Building a single model with high robustness against all file environments is difficult.

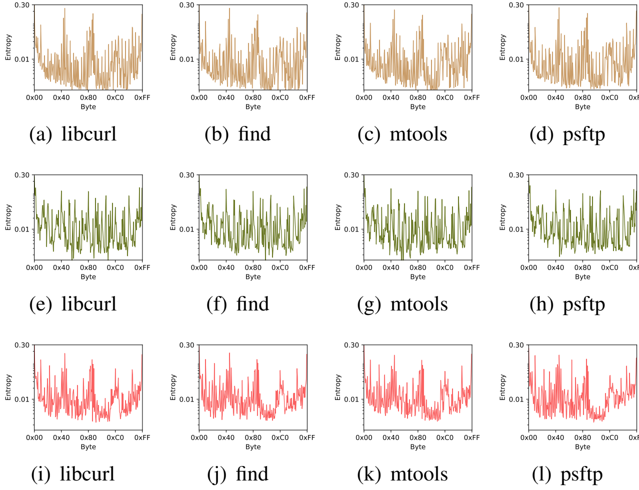
In response to this challenge, we propose a novel divide-and-conquer strategy. First, we split the embedding space of the mixed file environment into multiple embedding sub-spaces. Second, we choose one of the file environments $\mathcal{V}$ as the intermediate file environment, and divide the function similarity problem among N file environments with C_N^2 scenarios into $N-1$ sub-problems of function embedding transferring. Finally, we use trained adapters to transfer function embeddings from different file environments into the same file environment $\mathcal{V}$ for similarity calculation, which can alleviate the differences caused by different file environments. With the help of the divide-and-conquer strategy, our model generates robust function embeddings against different file environments.

As shown in Fig. 5, a similarity calculation problem with 2,556 scenarios is divided into 71 sub-problems, significantly reducing the problem's complexity. The distribution of function embeddings is different in different file environments. The adapters transfer function embeddings into the same file environment and keep similar functions clustered together. In this way, the function similarity can be quickly determined based on the distance in the embedding space. In this paper, we consider 3 architectures including x86, arm and mips, 2 word sizes including 32-bit and 64-bit, 2 compilers including Clang and GCC, and 6 optimization levels including O0, O1, O2, O3, Os and Ofast, which are a total of 72 file environments ($3 \times 2 \times 2 \times 6$). In binary code search, it is required that the embeddings of similar functions from any two file environments mentioned should be close, which is a complex problem with 2,556 scenarios (C_{72}^2). Our divide-and-conquer strategy turns this problem with 2,556 scenarios into 71 embedding-transferring problems. Considering that the number of O1 is in the middle of all optimization levels, We select the O1-GCC-x86-64 file environment as the intermediate file environment. Note that it can also choose other file environments as the intermediate file environment.

1) Entropy-based Binary Analysis: For the divide-and-conquer strategy, an important step is to identify the file environments. Architectures and word sizes of binary functions can be identified via their instructions. However, the problem is there is no information to directly indicate compilers and optimization levels in binary files.

To resolve this problem, we take an information-theoretic perspective to understand binaries and introduce entropy to identify compilers and optimization levels of binary files. Generally, code segments that have been compressed or encrypted tend to have higher entropy than native code [33]. This also can be applied in distinguishing different compilers and optimizations.

From the information-theoretic perspective, two binary code fragments, compiled by the same source code with O0 and O3 optimization levels respectively, carry the same code



(a) libcurl (b) find (c) mtools (d) psftp

(e) libcurl (f) find (g) mtools (h) psftp

(i) libcurl (j) find (k) mtools (l) psftp

Fig. 6. Entropy streams of binary files. The samples in the first, second, and third rows are from different file environments O0-Clang-x86-64, O1-GCC-arm-64, and O3-Clang-x86-64, respectively. Note the variation of the entropy streams between the file environments.

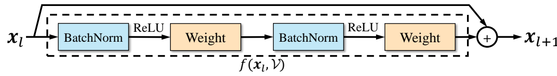


Fig. 7. The layer of ResNet

semantics. However, due to compiler optimization, the binary code fragment with `O3` is usually more refined than those with `O0`. Thus, the entropy of the binary code fragment with `O3` is different from those with `O0`. Fig. 6 displays the entropy streams from 12 different binaries from 3 file environments. The entropy stream of a binary file is computed by splitting its raw bytes into hexadecimal representation (`0x00`–`0xFF`). It can be observed that the entropy streams from the same file environments appear to be similar while distinct from those belonging to different file environments. Using entropy streams and the theory of entropy, we can identify different compilers and optimizations.

To prevent collision problems that single entropy streams might cause, we use the following features:

- The entropy stream of the `.text` segment, which are 256 respective probabilities of raw bytes (`0x00`-`0xFF`).
- The `.text` segment entropy which is the integral of the entropy stream over the `.text` section. This focuses on the executable part in binaries and avoids the impact of data segment changes.
- The file entropy is the integral of the entropy stream over the entire file, providing global information at the file level.

We use a Residual Neural Network (ResNet) [50] as the classifier, and the aforementioned features as inputs for identifying file environments. Fig. 7 shows the structure of the basic residual block. It consists of batch normalization and the linear transformation, and the activation function is `ReLU`. A skip connection using identity mapping from the input is added to the basic residual block output, which preserves the function semantics and helps tackle the vanishing gradient problem [50]. We sample inputs using a convolutional layer which converts the 258-dimension inputs into 64-dimension features. After 16 layers of residual basic blocks, we use linear prediction and a softmax function to do multi-class classification. Since `Os` and `Ofast` are incremental optimizations on `O2` and `O3`, respectively, we include `Os` and `Ofast` into `O2` and `O3`, respectively. Here, the softmax objective is 8 classes (2 compilers × 4 optimizations). The entropy analysis helps us identify file environments (compilers and optimization levels) in preparation for subsequent function embedding transferring.

2) Entropy-based Adapter layer: To calculate the similarity of binary functions from different file environments, we propose an entropy-based adapter layer after the IRFM. The entropy-based adapter layer acts as a mapping $\mathcal{F}$ to transfer function embeddings from different file environments into the same intermediate file environment $\mathcal{V}$ to alleviate the differences caused by different file environments. The mapping $\mathcal{F}$ should both preserve the function semantics and alleviate deviations due to the different file environments.

Given two function embeddings x and x' from the source code but different file environments $\mathcal{V}$ and $\mathcal{V}'$, we rewrite x as $x' - \alpha$ where α represents the bias between x and x'. Their similarity s is written as $s = \frac{1}{1+|(x'-\alpha)-x'|_2}$. When the bias α tends to 0, the similarity s tends to 1. So, the mapping $\mathcal{F}$ is a combination of the function embedding itself and the bias, where we fit the bias with $f(x, \mathcal{V})$. We use ResNets [50] to construct the mapping $\mathcal{F}$. The adapter weights are different according to the input file environment $\mathcal{V}$.

Training. To reduce the training complexity, we freeze the parameters of IRFM. We use the function similarity as the ground truth, i.e., dissimilar (0) and similar (1). The training objective is to make the similarity of similar functions approach 1, and the similarity of dissimilar functions approach 0. We use the Cross-Entropy loss as the loss function:

$$\mathcal{L} = -\sum_{i=0}^{N}\sum_{j=0}^{N}(y_{ij}\log(s_{ij}) + (1-y_{ij})\log(1-s_{ij})) \quad (9)$$

where y is the ground truth, s_{ij} is the similarity of functions i and j. We use the Adam optimizer to optimize parameters to minimize the loss $\mathcal{L}$. It is worth noting that when new compilers and optimization levels are introduced, we only need to train the corresponding mappings for the new file environments, which makes our approach more practical.

C. Progressive Search Strategy

Existing methods [10], [27], [35] use function embeddings to search for similar functions. This is a coarse-grained detection approach lacking fine-grained information (e.g., block-level features), which achieves low search overheads but leads to high false positives, especially for minor-change patched functions. While Marcelli *et al.* [34] using Graph Matching Networks [28] and other methods [58], [60] using Siamese network [3] to calculate the similarity of each function pair at the fine-grained level, which may achieve high performance but are computationally expensive.

Facing the complex vulnerability detection scenario, we propose a novel search strategy, named Progressive Search Strategy, to alleviate the computational burden while preserving a good performance and reducing false positives caused by patched functions in vulnerability detection. The strategy combines two sub-strategies. First, we use function embeddings as global summaries for coarse-grained search. Second,

we design a pairwise similarity calibration for candidate functions to supplement the function embeddings with fine-grained information to keep high precision for vulnerability detection.

1) Function Embedding Search: With the help of the entropy-based adapter, function embeddings generated by IRFM for similar functions are close in the embedding space. To efficiently detect similar functions in a large function repository, we use Euclidean distance similarity of function embeddings to retrieve candidates in a coarse-grained manner, which will significantly reduce the scope of fine-grained detection and alleviate the computational burden.

In Algorithm 1 Ln.2, we perform matrix computations on pre-generated function embeddings to obtain the Euclidean distance similarity. Given a query set containing n functions and a function repository containing m functions, we use VulHawk to generate their function embedding matrices $N \in \mathbb{R}^{n\times d}$ and $M \in \mathbb{R}^{m\times d}$. The matrix computation function is as follows:

$$\begin{aligned} Y &= (N * N)\vec{1_d}\vec{1_m}^T + \vec{1_n}((M * M)\cdot \vec{1_d})^T - 2NM^T \\ &= (N * N)\vec{1_d}\vec{1_m}^T + \vec{1_n}\vec{1_d}^T(M * M) - 2NM^T \end{aligned} \quad (10)$$

where $\vec{1_d} \in \mathbb{R}^{d\times 1}$ denotes the calculation of the sum of each row values, and $\vec{1_m}^T \in \mathbb{R}^{1\times m}$ is used to expand a column vector into a matrix. The result $Y \in \mathbb{R}^{n\times m}$ represents the distances between the query set and the function repository, where y_{ij} represents the distance between i-th function in the query set and j-th function in the function repository. Then, we convert Euclidean distance matrix Y to similarity S ($s_{ij} \in [0, 1]$) using the Equation 7. In Algorithm 1 Ln.4, with the default threshold h, we take the top-K candidates as the results. Note that it can also use libraries such as Faiss [21] for further search acceleration.

2) Similarity Calibration: As with other methods, our function embeddings are designed to be robust against scenarios that cross different file environments. They are tolerant to minor changes. At the same time, they may be insensitive to small vulnerability patches due to the lack of fine-grained information. This will bring intolerable false positives and increases the analysis burden on researchers. For high-precision binary code search, we propose a similarity calibration for fined-grained detection. It combines the information of basic blocks, string constants, and imported functions to compute pairwise similarity scores, from which the vectors are extracted and combined with the function-level information to boost the performance of vulnerability detection.

Block-level Features. The block-level features, such as the block embedding distribution and function size, may be lost by function-level embeddings. In many cases, the differences between functions lie in small substructures and are hard to be reflected by function embeddings. An analogy is that, in graph matching, the graph matching performance based on graph embeddings can be enhanced via fine-grained node-level information [1]. In Algorithm 1 Ln.6, we calculate the basic block similarities between the queried function and its candidate functions to supplement with block-level information. Given two functions f_1 and f_2, we first use Equation 10 to calculate the similarity of their basic block sets, and then count the maximum similarity according to the range [0, 0.2), [0.2, 0.4), [0.4, 0.6), [0.6, 0.8) and [0.8, 1.0]. In this way, we generate

Algorithm 1: Progressive Search Strategy Algorithm

Input: A set of queried functions $\mathcal{F}_{query}$ and the function repository $\mathcal{F}$.
Output: The search result $S_{matched}$.

```
1  S_matched = { } ;
   /* Function search */
2  M = EuclideanSimilarity(F_query, F) ;
   /* Similarity calibration */
3  for each f_query in M do
4      F_candidates = F →GetCandidates(f_query, M, topK, h) ;
5      for each f_candidate in F_candidates do
6          V_blk = GetBlockSimilarity(f_query, f_candidate) ;
7          V_str = GetStringSimilarity(f_query, f_candidate) ;
8          V_imp = ImportsJacarrdSimilarity(f_query, f_candidate) ;
9          V = concatenate(V_blk, V_str, V_imp, s_func) ;
10         s' = SimilarityCalibration(V) ;
11         if s' > h then
12             S_matched = S_matched ∪ (f_query, f_candidate) ;
13 Output S_matched;
```

a 5-dimension vector V_{blk} as supplements at the basic block level.

String Features. Since the string constants and imported functions are the same or similar in similar function pairs, their similarity also plays a role in indicating the function similarity. Algorithm 1 Ln.7 is to calculate the string similarity. We use a pre-trained model Sentence-BERT [46] to generate string embeddings, which makes our similarity calculation more powerful because it could allocate larger similarity scores to similar strings. For string constants, we use Sentence-BERT to generate embeddings for concatenated strings. Following the Sentence-BERT settings [46], we use the cosine similarity of string embeddings. Here, we generate a 2-dimension vector V_{str}: the string similarity and the total length of strings.

Imported Functions. In Algorithm 1 Ln.8, we use the Jaccard Index to calculate the similarity s_i of two imported function sets I_1 and I_2: $s_i = 1 - \frac{|I_1 \cap I_2|}{|I_1 \cup I_2|}$, where $|I_1 \cap I_2|$ denotes the number of imported functions in both sets, and $|I_1 \cup I_2|$ denotes the total number of imported functions. Here, we generate a 2-dimension vector V_{imp}: the similarity of imported functions and the total number of imported functions.

After computing the above three vectors, we concatenate these vectors and the similarity s from the function embedding search into the vector V. Then, we feed the vector V into a feed-forward network to learn weights and predict the final function similarity s'. We use the Cross-Entropy loss function to optimize the network weights. Finally, we use the default threshold h to filter out similar functions as the results.

IV. Evaluation

In this section, we compare VulHawk with available state-of-the-art binary code search approaches and binary code similarity detection approaches in three different scenarios: cross-compilers, cross-optimization levels, and cross-architectures. The experiments aim to answer the following research questions: **RQ.1** Given two binary functions, can VulHawk determine whether they are similar (i.e., one-to-one comparison)? **RQ.2** Can VulHawk be used for searching one function in a large function repository (i.e., one-to-many search)? **RQ.3** Can VulHawk identify how many functions are similar from two binaries (i.e., many-to-many matching)? **RQ.4** How efficient

TABLE II. THE SETTING OF SEVEN EVALUATION TASKS.

Task	XO	XA	XC	XO+XA	XO+XC	XA+XC	XO+XA+XC
compiler	×	×	◯	×	◯	◯	◯
architecture	×	◯	×	◯	×	◯	◯
optimization	◯	×	×	◯	◯	×	◯

◯ represents the function pairs with different settings for this, while × represents the function pairs with the same settings for this.

is VulHawk in searching in the large repository (i.e., runtime efficiency)? **RQ.5** How is the contribution of each component in VulHawk (i.e., ablation study)? **RQ.6** Can VulHawk detect 1-day vulnerabilities in the real world?

A. Implementation and Experiment Setup

We implement VulHawk using Transformers [52], NetworkX [17], and PyG [51], based on Python 3.8.5. The disassembler is IDA Pro 7.5 [45]. Our experiments are performed on a desktop computer running Windows 10 with Intel Core i9-10920X CPU, 64GB DDR4 RAM, and one NVIDIA RTX3090 GPU. To facilitate further research, we make the programs and datasets publicly available on GitHub.

Hyper-params. In the RoBERTa model, $layer$=6, $head$=8 and $hidden_dimension$=256; in the GCN model, $hidden_dimension$=256 and $layer$=16; in the model to identify file environments, $layer$=8, $input_dimension$=258, $hidden_dimension$=64; in the entropy-based adapter, $layer$=2 and $hidden_dimension$=256; in the calibration networks, $input_dimension$=10 and $layer$=3. These settings are a trade-off between efficiency and performance.

1) Datasets: As with the previous work [10], [11], [27], [30], [34], [35], [55], we use following projects Coreutils-8.30, Curl-7.70, Diffutils-3.6, Findutils-4.7.0, Libmicrohttpd-0.9.75, mtools-4.0.36, OpenSSL-1.1.1l, putty-0.74, wget2-2.0.0, and sqlite-3.37.1. The dataset is widely used in practice and related work. We compile these projects using 2 compilers (`GCC-10` and `Clang-10`), 6 optimization levels (`O0`, `O1`, `O2`, `O3`, `Os` and `Ofast`), 3 architectures (`x86`, `arm` and `mips`), and 2 word sizes (`32-bit` and `64-bit`). In total, we obtain 3,393 files containing 596,099 binary functions and 13,398,845 basic blocks. To evaluate the generalization capability of the trained model on unseen binaries, we randomly select half of the binaries in our dataset for model training. Any binaries in the training set do not appear in the evaluation set, which ensures that the evaluation set is unknown to VulHawk. These settings ensure that generated binaries are more similar to real-world cases, making the evaluation practical and representative. As Table II shown, we identify seven different tasks to evaluate: (1) XO, (2) XC, (3) XA, (4) XO+XC, (5) XO+XA, (6) XC+XA, and (7) XO+XC+XA, where ×/◯ represents the function pairs have the same/different settings for this. For example, the task XO+XC represents the function pairs with different optimizations and compilers but the same architectures.

2) Baselines: In the comparative experiments, we select the state-of-the-art methods as baseline techniques:

- **PalmTree** [27], the state-of-the-art BCSD approach, uses pre-trained models to generate instruction embeddings which can be used to calculate function similarity.
- **SAFE** [35] uses a word2vec model to generate instruction embeddings, and proposes a recurrent neural network to generate function embeddings.
- **Trex** [41], the state-of-the-art BCSD method, uses transfer-learning-based models based on micro-traces to generate function embeddings for matching similar functions.
- **Asteria** [58] uses a Tree-LSTM network based on abstract syntax trees (ASTs) and adopts a Siamese network [3] to calculate function similarities, where its experiments show Asteria outperforms Gemini [55].
- **Asm2Vec** [10] uses an unsupervised learning model to generate function embeddings using the PV-DM model.
- **BinDiff** [61], a state-of-the-art commercial BCSD tool, uses multiple features to perform similar function detection. We measure it with its latest version 7 using default settings.
- **Graph Matching Networks (GMN)** [28]. The existing studies [34] show that GMN based on CFG has natural advantages in cross-architecture scenarios. GMN is set up as the study [34].

For the above baselines, we use their original implementations and default settings. To evaluate the contributions of the entropy-based adapter and the similarity calibration strategy, we set up three configurations:

- **VulHawk**: the original VulHawk.
- **VulHawk-ES**: VulHawk replaces the entropy-based adapter with neural networks and does not use the similarity calibration.
- **VulHawk-S**: VulHawk without the similarity calibration.

B. One-to-one Comparison

We benchmark the performance of VulHawk and baselines with one-to-one function similarity detection, which is widely performed in previous methods [27], [34], [41], [55]. As in their experiment settings, we construct a balanced evaluation set of 50k positive function pairs and 50k negative function pairs, and an unbalanced evaluation set of 1,400 positive function pairs and 140k negative function pairs for each task. We use the area under curve (AUC) of the receiver operating characteristic (ROC) curve as measurements. AUC is a comprehensive performance metric of a model integrating all the possible classification thresholds. Table III shows the comparative results of VulHawk and other baselines.

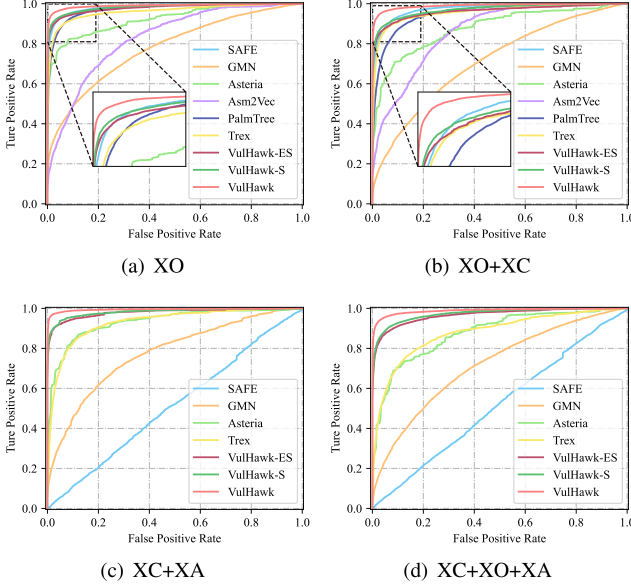
As shown, VulHawk outperforms SAFE, Asteria, GMN, PalmTree, Asm2Vec, and Trex in terms of AUC scores on both balanced and unbalanced sets in all experiment settings. For example, in the cross-architecture (XA) experiment VulHawk achieves an AUC of 0.998, where Trex gets an AUC of 0.947, Asteria gets an AUC of 0.951, SAFE only gets an AUC of 0.509, and PalmTree and Asm2Vec fails in the cross-architecture experiment. PalmTree and Asm2Vec focus on a mono-instruction set (`x86`), which cannot deal with functions from different architectures. Though SAFE trained its model on the different instruction sets, it is still hard to build semantic relations between instructions from different ISAs and embed similar functions from different architectures into similar vectors. As stated in its Github issues[2], current SAFE hardly supports cross-architecture tasks. Though GMN

[2]https://github.com/gadiluna/SAFE/issues/4

TABLE III. AUC SCORE OF ONE-TO-ONE EXPERIMENTS

	Balanced Set							Unbalanced Set						
	XC	XO	XA	XC+XO	XO+XA	XC+XA	XC+XO+XA	XC	XO	XA	XC+XO	XO+XA	XC+XA	XC+XO+XA
Asm2Vec*	0.796	0.854	-	0.861	-	-	-	0.803	0.830	-	0.864	-	-	-
Asteria	0.904	0.924	0.951	0.879	0.950	0.933	0.877	0.905	0.933	0.956	0.870	0.950	0.935	0.892
PalmTree*	0.965	0.973	-	0.952	-	-	-	0.965	0.969	-	0.948	-	-	-
GMN	0.769	0.780	0.865	0.711	0.726	0.775	0.717	0.773	0.783	0.870	0.718	0.740	0.775	0.723
SAFE	0.980	0.983	0.509	0.975	0.505	0.513	0.515	0.979	0.984	0.504	0.975	0.500	0.512	0.509
Trex	0.981	0.965	0.947	0.963	0.901	0.928	0.883	0.984	0.962	0.946	0.957	0.896	0.937	0.879
VulHawk	**0.993**	**0.990**	**0.998**	**0.990**	**0.992**	**0.994**	**0.988**	**0.996**	**0.988**	**0.998**	**0.991**	**0.993**	**0.995**	**0.987**
VulHawk-ES	0.971	0.979	0.989	0.962	0.9633	0.979	0.966	0.974	0.979	0.987	0.963	0.966	0.980	0.961
VulHawk-S	0.978	0.983	0.992	0.971	0.972	0.983	0.973	0.980	0.985	0.990	0.971	0.974	0.982	0.968

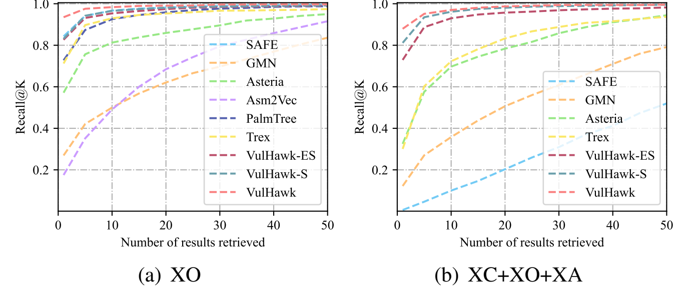
* PalmTree and Asm2Vec do not support cross-architecture tasks.



(a) XO (b) XO+XC

(c) XC+XA (d) XC+XO+XA

Fig. 8. ROC curves of one-to-one function comparison on the balanced evaluation set.



(a) XO (b) XC+XO+XA

Fig. 9. Comparison of the recall at different K values for XO and XC+XO+XA tasks.

achieves an AUC of 0.865 in the XA task, it performs poorly in other tasks. Because CFGs are robust against architectures but changed in different compilers and optimization levels. VulHawk lifts binary code into microcode to mitigate the problem of different ISAs and uses entropy-based adapters to transfer embeddings into the intermediate file environment, which makes VulHawk outperform other baselines. Fig. 8 presents detailed ROC curves of VulHawk and other baselines, where the closer the ROC curve is to the upper left, the better the performance. As shown in Fig. 8, the more differences in the file environment (compiler, architecture, and optimization levels) between functions, the worse the performance of function comparisons. Fortunately, with the help of the entropy-based adapter and similarity calibration, VulHawk is resilient to these differences and achieves high performance. **Answer to RQ.1**: VulHawk ranks the first in all one-to-one comparison tasks, demonstrating the robustness of VulHawk using divide-and-conquer strategy against cross-architecture, cross-optimization level, and cross-compiler tasks.

C. One-to-many Search

In this section, we evaluate the performance of the one-to-many search. As in the study [34], we use ranking measurements to evaluate the model performances in searching applications, e.g., the vulnerability search, which retrieves candidate functions from a large database. We use the recall (Recall@K) at different K thresholds as the metrics, which is a widely used ranking metric. We use the unbalanced set (in Section IV-B) as the evaluation set. In the evaluation, the models calculate and rank the similarity between each queried function and its positive/negative samples.

We collect recall at different top-K results and plot recall against k in Fig. 9. Results show VulHawk outperforms the state-of-the-art approaches and achieves the best recall@1 of 0.935 in the XO task and 0.879 in the XC+XO+XA task. In the XC+XO+XA task, when the number of results retrieved is over 30, the recall of each approach tends to be stable, where VulHawk achieves the recall@30 around 0.994, VulHawk-ES achieves the recall@30 around 0.968, VulHawk-S achieves the recall@30 around 0.988, Trex achieves the recall@30 around 0.888, and SAFE gets the recall@30 around 0.310. The recall@K of SAFE in the XC+XO+XA task is close to the random probability ($\frac{K}{100}$), because SAFE is not robust on cross-architecture tasks due to heavy OOV issues, which has been shown in the one-to-one comparison. In the one-to-many search against a large size of function repository, the weakness of SAFE is magnified, so it only obtains a recall@1 of 0.007. **Answer to RQ.2**: VulHawk can retrieve the best candidates accurately in a large function repository.

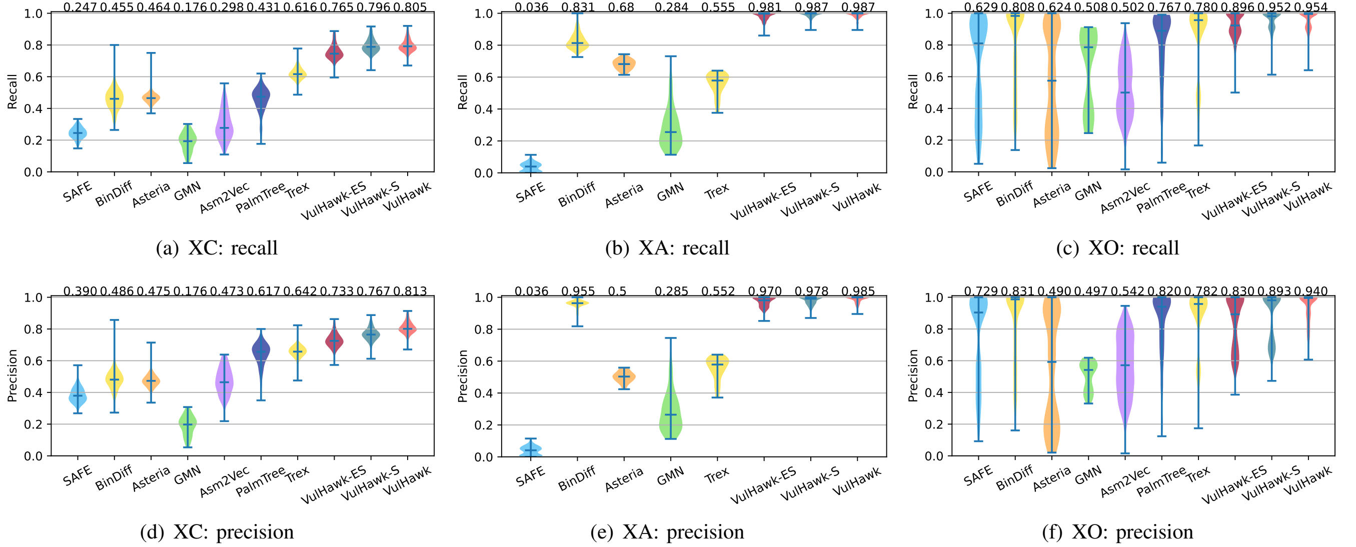
D. Many-to-many Matching

We conduct experiments to measure the performance on many-to-many matching, which is performed in previous work [10], [61]. Many-to-many matching is used to measure the similarity of two given binaries at the function level. As their experiment settings, we construct an evaluation set of binary pairs for each task, and each tool to generate the best function similarity matching for each binary pair. We report each tool's

TABLE IV. RESULTS OF MANY-TO-MANY MATCHING

	Recall								Precision							
	XO					XC	XA	Average	XO					XC	XA	Average
	O0-O3	O2-O3	Ofast-O3	Ofast-Os	Average				O0-O3	O2-O3	Ofast-O3	Ofast-Os	Average			
SAFE	0.227	0.823	0.946	0.519	0.629	0.247	0.036	0.304	0.315	0.918	0.975	0.709	0.729	0.390	0.036	0.385
Asteria	0.299	0.801	0.869	0.527	0.624	0.464	0.680	0.589	0.372	0.558	0.580	0.451	0.490	0.475	0.500	0.488
Asm2Vec*	0.420	0.736	0.400	0.452	0.502	0.298	-	0.400**	0.503	0.804	0.402	0.457	0.542	0.473	-	0.508**
BinDiff	0.365	0.979	**0.994**	0.892	0.808	0.455	0.831	0.480	0.419	0.981	**0.994**	0.925	0.831	0.486	0.955	0.757
PalmTree*	0.414	0.926	0.944	0.784	0.767	0.431	-	0.599**	0.487	0.962	0.976	0.856	0.820	0.617	-	0.718**
GMN	0.141	0.576	0.908	0.409	0.508	0.176	0.284	0.323	0.104	0.588	0.908	0.386	0.497	0.176	0.285	0.319
Trex	0.432	0.950	0.954	0.784	0.780	0.616	0.555	0.650	0.481	0.948	0.955	0.743	0.782	0.642	0.552	0.659
VulHawk	**0.876**	**0.994**	**0.994**	0.950	**0.954**	**0.805**	**0.987**	**0.915**	**0.818**	**0.995**	**0.994**	**0.950**	**0.940**	**0.813**	**0.985**	**0.913**
VulHawk-ES	0.793	0.910	**0.994**	0.887	0.896	0.765	0.981	0.881	0.593	0.909	0.992	0.824	0.830	0.733	0.970	0.844
VulHawk-S	0.873	0.984	**0.994**	**0.958**	0.952	0.796	**0.987**	0.912	0.673	0.984	0.992	0.923	0.893	0.767	0.978	0.879

* PalmTree and Asm2Vec do not support cross-architecture tasks.
** Average of results cross compiler tasks and cross-optimization levels tasks, without cross-architecture tasks.



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

(d) XC: precision (e) XA: precision (f) XO: precision

Fig. 10. The violin plots of many-to-many matching.

average recall and precision results under different tasks in Table IV.

Among them, baselines have the lowest result in the O0-O3 of the XO experiment. VulHawk achieves a recall of 0.876 in this experiment, which improves the recall by 385.9%, 292.9%, 208.6%, 240.0%, 211.6%, 621.3% and 202.8% over SAFE, Asteria, Asm2Vec, BinDiff, PalmTree, GMN, and Trex. Interestingly, the worst results of VulHawk (0.805) are in the XC experiment, not the O0-O3 experiments. Nevertheless, VulHawk still outperforms the state-of-the-art methods in the XC experiments. In the `O3` option, to reduce the size and speed up the efficiency of binary functions, it compresses redundant instructions in the `O0` option into concise instructions, which strengthens the main semantics and removes redundant semantics of functions, leading to semantic differences. VulHawk uses instruction simplification to distill binary functions and remains the main semantics, which helps VulHawk mitigate the impact of optimization levels and make it resilient to cross-optimization level experiments.

Fig. 10 shows the distribution of recall and precision in XC, XA, and XO tasks with violin plots, where we annotate the average result of each method above the figure. Compared to SAFE, BinDiff, Asteria, Asm2vec, PalmTree, GMN, and Trex, VulHawk's recall and precision probability distribution is closer to 1 and more concentrated, while the distribution of results for other methods is scattered and unstable against different scenarios. This shows that the performance of VulHawk is better and more stable than other baselines.

Compared with one-to-one comparison, the recall and precision rates of the many-to-many matching are lower, because there are more negative samples in many-to-many matching, and wrong-matched pairs impact subsequent results of the matching algorithm (e.g., Hungarian algorithm [25]). **Answer to RQ.3**: VulHawk can be used to match similar functions between two binaries, and it outperforms the state-of-the-art methods in many-to-many matching.

E. Runtime Efficiency

We evaluate the runtime efficiencies of VulHawk with different settings. Given a function, we use the model to be evaluated to extract features and generate its function embedding, and then retrieve top-10 candidate functions from the repository. The record time is from extracting features to returning the similarity of candidate functions, and each test is measured ten times to minimize accidental factors. The repository size is set to 10^3, 10^4, 10^5, and 10^6.

Table V shows the time cost of searching a function in the repository of different sizes and their throughput. The throughput represents the number of repository functions that

TABLE V. EFFICIENCY OF VULHAWK AND BASELINES

	$1{:}10^3$	$1{:}10^4$	$1{:}10^5$	$1{:}10^6$	Throughput
VulHawk-ES	0.255s	0.272s	0.434s	1.948s	590,091
VulHawk-S	0.266s	0.276s	0.425s	1.997s	577,251 (-2.2%)
VulHawk	0.286s	0.345s	0.499s	2.070s	559,898 (-5.1%)

can be retrieved in one second, which does not consider the embedding generation's overhead. The results show that VulHawk is slower than VulHawk-ES and VulHawk-S, because VulHawk uses the entropy-based adapter during the embedding generation and uses the similarity calibration during the search process. It is acceptable to trade a small overhead for higher precision and recall. With the GPU acceleration, VulHawk can search one function from 10^6 functions in about 2 seconds. **Answer to RQ.4**: VulHawk can maintain high efficiency in the large function repository (10^6).

F. Ablation Study

We analyze the contributions of the entropy-based adapter and the similarity calibration strategy in VulHawk.

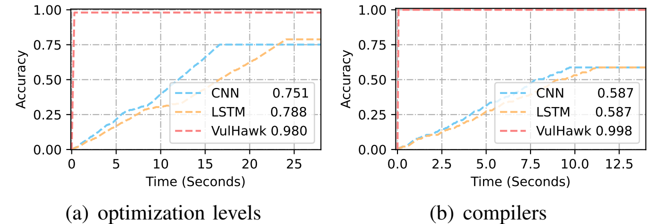
Entropy-based Adapter. As shown in Table III, VulHawk-S has a higher AUC than VulHawk-ES in seven one-to-one function comparison tasks. In Fig. 9, VulHawk-S achieves higher recalls@K than VulHawk-ES in the one-to-many search. These show that VulHawk-S outperforms VulHawk-ES, which turns out that the performance boost is from the contribution of the entropy-based adapter, not the extra neural networks. The entropy-based adapter transfers function embeddings from different file environments into the same file environment, alleviating the differences caused by compilers and optimization levels. For example, in the many-to-many matching scenarios, VulHawk-ES get a recall of 0.793 in the O0-O3 experiment, while VulHawk-S, with the help of the entropy-based adapter, alleviates the differences caused by different file environments, reaching a recall of 0.873.

Similarity Calibration. Table III and Fig. 9 show the VulHawk achieves better results than VulHawk-S in one-to-one function comparison and one-to-many search scenarios. Table IV also shows VulHawk is more precise than VulHawk-S in many-to-many matching scenarios. The precision of VulHawk improves from 0.593 to 0.818 in the O0-O3 task of many-to-many matching, because VulHawk uses the similarity calibration to supplement the function similarity from block-level features, string features, and imported functions, which makes the information considered in the similarity calculation more comprehensive and improve detection precision.

Training Tasks. To evaluate the contributions of three training tasks, we also evaluate VulHawk with different training settings. To more clearly measure the contribution of training tasks, the evaluation models do not use entropy-based adapters and the similarity calibration, and differ only on training tasks. In the XC+XO+XA task of the one-to-one comparison scenario, the model trained by the MLM task gets an AUC of 0.833, the model trained by the MLM+ROP task gets an AUC of 0.934, and the model trained by MLM+ROP+ABP achieves an AUC of 0.966. The ROP training task helps the model to learn root token semantics, which can use root token semantics to replace OOV operands for alleviating OOV issues. The ABP training task helps the model learn the order-relations of microcode. Both of them improve the model performance to distinguish function similarities. **Answer to RQ.5**: The entropy-based adapter, the similarity calibration, and ROP and ABP training tasks have positive contributions to the performance of VulHawk.

G. File Environment Identification

We also evaluate the accuracy of the entropy-based file environment identification. Here, we use 10-fold cross-validation to split all binaries for training and evaluation, as in traditional machine learning settings. These binaries are from various architectures (`x86`, `arm`, and `mips`) and different compilers (`GCC` and `Clang`). Pizzolotto *et al.* [44] use CNN models and LSTM models on function bytes to identify file environments. To better demonstrate our method's performance, we download their pre-trained models and set them (CNN and LSTM) as comparisons. Note that compiler and optimization levels of given binaries are unknown and varied in practice, so we do not fix other parameters (e.g., architectures and optimization levels) when evaluating one parameter (e.g., compilers) to ensure practicability.



(a) optimization levels (b) compilers

Fig. 11. Comparison with existing methods on file environment identification.

Fig. 11 shows the trend of the accuracy of VulHawk and other methods over time. There are the following observations from the results: (1) VulHawk outperforms CNN and LSTM in both identifications of compiles and optimization levels. (2) The identification speed of VulHawk is faster than CNN's and LSTM's. (3) The accuracy curve of VulHawk is steady, while the curves of CNN and LSTM are fluctuating. It indicates that VulHawk is more generalized in various scenarios, while CNN and LSTM are struck in some cases. We conduct an in-depth analysis as follows. Compared with CNN and LSTM, VulHawk adopts entropy streams including the global information of binaries, instead of a single function's raw bytes, which makes the accidental deviation of functions not impact VulHawk much. The entropy keeps obvious discrimination in different file environments, while the difference in raw bytes is not obvious. These make VulHawk keep high generalization and outperform CNN and LSTM. Besides, VulHawk outperforms CNN and LSTM not only in accuracy but also in efficiency. Because VulHawk uses a powerful feature (entropy) and our model is much lighter than CNN and LSTM, e.g., the size of parameters in our model is only 1.3% of that of CNN.

To comprehensively evaluate our entropy-based file environment identification, we conduct experiments in various scenarios with different architectures (`x86`, `mips`, and `arm`), file sizes (small sizes and large sizes), and file types (libraries and executable files). Here, considering the file size distribution, we treat file sizes less than 1024 `KB` as small files and vice versa as large files. Fig. 12 shows the results, and each block is annotated with the proportion of classification. For example, in

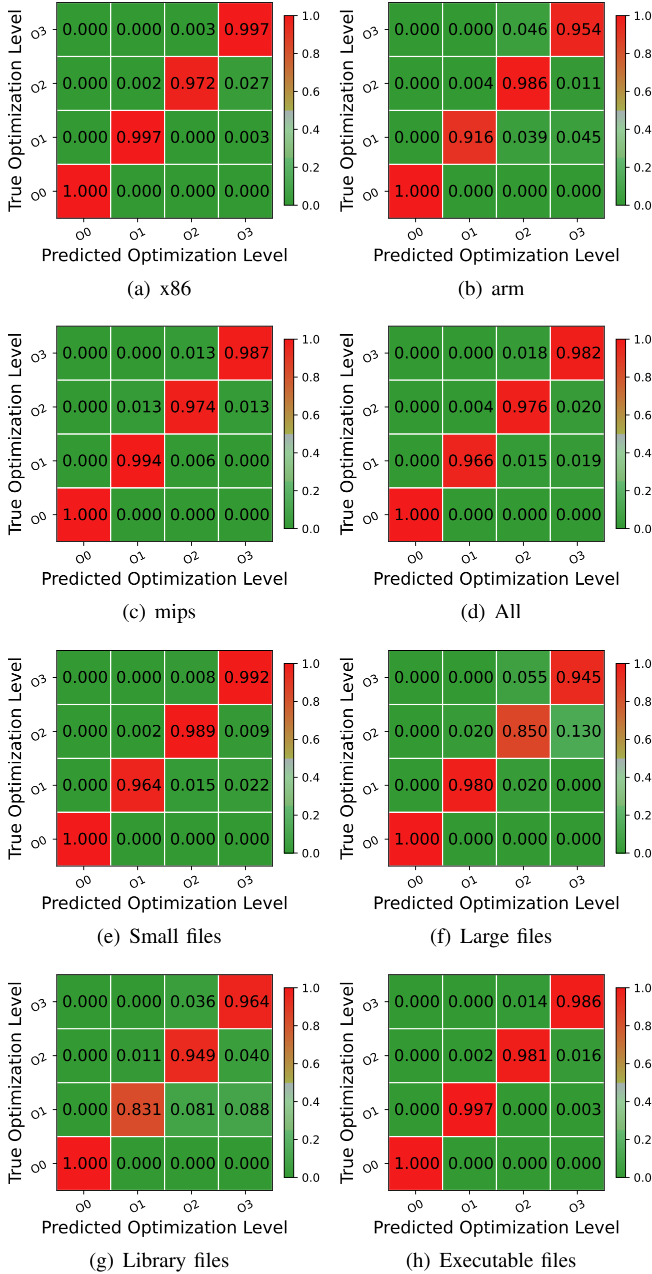


Fig. 12. The results of file environment identification. The overall optimization accuracy is 0.980.

arm-O3 predictions, 95.4% are correctly classified as O3, and 4.6% are classified as O2, where the overall accuracy is 96.2%. The option O0 achieves a high accuracy, because O0 uses default optimizations to reduce the compilation time, which is significantly different from other options. The identifications of O2 and O3 in the mips and library files achieve the worst results, where most misidentifications are centered on the distinction between O2 and O3. Compared with O0, the O3 option adds more than 285 optimizations, while O3 only has 3 more optimizations than O2. Thus, binaries from O2 and O3 have highly similar structures. Since binaries from O2 and O3 are highly similar, these misidentifications are acceptable and have little impact on the binary function search. When we consider O2 and O3 together, the accuracy of distinguishing them from O0 and O1 is over 90% in all scenarios. The results show our entropy-based file environment identification is powerful in various scenarios with different architectures, file sizes, and file types.

TABLE VI. RESULTS OF VULNERABILITY SEARCH AND CONFIRMED VULNERABLE FUNCTIONS

#	CVE	Confirmed #	VulHawk	Trex	SAFE	GMN	Asteria
1	2015-0286	3	3;0;0*	0;0;3	0;0;3	3;215;0	3;0;0
2	2015-1789	3	3;0;0	0;0;3	0;0;3	2;766;1	3;0;0
3	2016-0797	8	8;0;0	0;0;8	0;0;8	8;2073;0	8;0;0
4	2016-0798	4	4;0;0	0;0;4	0;0;4	4;287;0	4;0;0
5	2016-2176	4	4;0;0	0;0;4	3;0;1	4;335;0	4;0;0
6	2016-2182	14	14;0;0	9;0;5	12;0;2	3;7814;11	14;0;0
7	2016-6303	17	17;0;0	13;0;4	17;4459;0	17;1802;0	17;0;0
8	2016-8618	10	10;0;0	9;0;1	9;0;1	9;6520;1	10;0;0
9	2016-8622	10	10;0;0	9;0;1	10;753;0	10;9084;0	10;4;0
10	2018-1000301	9	9;0;0	4;0;5	9;0;0	9;4801;0	9;27;0
11	2021-22924	10	10;0;0	4;0;6	9;0;1	10;2264;0	10;2;0
12	2021-23840	1	1;0;0	1;0;0	0;0;1	1;381;0	1;19;0
	Total	93	**93;0;0**	49;0;44	69;5212;24	80;36342;13	93;52;0

* "3;0;0" represents VulHawk detects three true positives, zero false positives, and zero false negatives.

H. 1-day Vulnerability Detection from Firmware

In this experiment, we collect 20 of the latest IoT firmware images from three vendors (D-Link, TP-Link, and NetGear) and perform VulHawk and other baselines in the 1-day vulnerability detection task. The projects OpenSSL and Curl are widely used in IoT firmware, so we select them as the targets and build a vulnerability repository based on the Common Vulnerabilities and Exposures (CVE) database. The repository contains vulnerable functions and their patched functions of 12 relevant CVEs, where their details and ground truth are shown in Table VI. In total, there are 53,739 functions, including 93 related vulnerable functions and 119 related patched functions. For each vulnerable/patched function, we use VulHawk to generate their function embeddings and record their fine-grained features for similarity calibration. In the vulnerability detection phase, we use all functions in the firmware libraries as function queries and perform a one-to-many search in the built vulnerability repository.

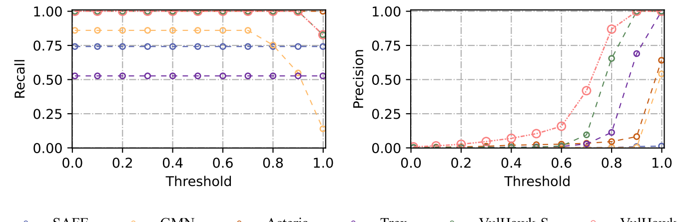


Fig. 13. Baseline comparison for 1-day vulnerability detection with different thresholds.

Table VI shows the results of VulHawk and other baselines with their best threshold according to Fig. 13. For 12 CVEs, Trex has zero false positives but 52.7% recall; GMN achieves 86.0% recall but brings 36,342 false positives; Asteria achieves 64.1% precision with zero false negatives; while VulHawk achieves the best performance with zero false positives and 100% recall. Furthermore, we make the following observations. (1) VulHawk outperforms baselines in 1-day vulnerability detection with high recall and precision, which frees researchers from the burden of detecting known vulnerabilities

in unknown binaries. (2) The impact of third-party library vulnerabilities in IoT firmware is still serious, and known vulnerabilities in firmware images have not been fixed in time. Even the latest firmware still contains vulnerabilities from 7 years ago. (3) The firmware images from adjacent models of the same vendor generally have similar vulnerable functions (see Appendix B). (4) Detecting vulnerabilities only by third-party library versions will lead to false positives, because third-party libraries may not include specific vulnerable functions due to the lack of corresponding modules. For example, TP-LINK Deco M4 uses OpenSSL 1.0.2d, which is an affected version of CVE-2016-6303, but it is not affected by this vulnerability because it does not have the corresponding module.

False Positive Analysis. We change the thresholds to analyze the impact of different thresholds on 1-day vulnerability detection. Fig. 13 shows the curves of recall/precision with different thresholds. As the threshold increases, the recall of VulHawk decreases slightly, while its precision increases significantly. To support XC, XO, and XA tasks, SAFE, GMN, and Asteria design their models to be robust against minor changes of functions, which leads to difficulty in distinguishing different functions and introduces false positives. Trex uses fine-grained features as micro-traces to detect vulnerable functions, which achieves high precision but suffers from false negatives. We propose a progressive search strategy combining coarse-grained search and fine-grained similarity calibration, which allows VulHawk to achieve high recall while maintaining high precision. For example, when the threshold is set to 0.9, VulHawk reaches 100% recall with zero false positives, which outperforms other baselines. We introduce VulHawk-S, which does not use similarity calibration. Compared with VulHawk-S, VulHawk shows a higher precision at the same threshold in 1-day vulnerability detection. Because fixed-length embeddings are designed to be robust against minor changes introduced by different file environments, which are coarse for precise vulnerability detection. We propose a similarity calibration to supplement function embeddings with fine-grained features to boost the detection precision. The patched functions with minor modifications may bring false positives when the threshold is set small. For example, the function `sub_20cc8` in `libcurl.so` of NetGear RBR20 is a patched function of CVE-2016-8618. Since this patch is a minor change and does not modify any values[3], VulHawk will misidentify `sub_20cc8` as a vulnerable function when the threshold is below 0.85. Fortunately, when the threshold of VulHawk is set to 0.9, these false positives are significantly reduced with 100% recall.

Furthermore, we use VulHawk to perform vulnerability detection in the latest projects, and find a suspected stack overflow vulnerability in `OpenSSL-3.1.0/ssl/ssl_lib.c`, which has been fixed as a result of our report. This shows VulHawk's capability of detecting new vulnerabilities. The details are shown in Appendix B. Finally, we have safely reported the discovered vulnerabilities in our experiments to the corresponding vendors and developers. **Answer to RQ.6**: VulHawk can distinguish vulnerable functions and their patched functions and detect 1-day vulnerabilities with high performance over the state-of-the-art methods in the real world.

[3] https://curl.se/CVE-2016-8618.patch

V. Discussion

In this section, we discuss the divide-and-conquer strategy, the soundness of IRFM, and future research.

From the results, these baseline approaches are difficult to achieve high performances in all three tasks of cross-compilers, cross-optimization levels, and cross-architectures. This is because the differences introduced by compilers, architectures, and optimization levels are different. So it is challenging to build a robust model against architectures, compilers, and optimization levels at the same time. While VulHawk uses the entropy-based adapter to implement the divide-and-conquer strategy, which divides the similarity calculation problem of C_N^2 cross-file-environment scenarios into $N-1$ embedding transferring sub-problems. By transferring function embeddings from different file environments to the same file environment, VulHawk can alleviate the differences caused by compilers, architectures, and optimizations. It can be found that the greater the differences introduced by the file environments, the improvements are more obvious brought by the entropy-based adapter (e.g., the XC task in Table IV). So the divide-and-conquer strategy is very effective in the complex IoT firmware search scenario.

The IRFM aims to generate robust basic block embeddings and function embeddings, which are crucial for subsequent detection and search. Here, we discuss its soundness: (1) In VulHawk, we use microcode, a well-established intermediate representation, to mitigate the differences caused by instruction set architectures. (2) To increase the main semantics' weights, we simplify the redundant and obfuscated instructions, where instructions to be removed are strictly filtered to avoid accidental deletion of return values, global variables, and arguments to subfunctions (see Section III-A2). The experiment results show that instruction simplification makes the distance between similar blocks closer and draws dissimilar blocks further (Appendix A). For example, the average distance between similar blocks reduces by 44.4%, and the average distance between dissimilar blocks increases by 2.6% in the XC task. (3) Applying NLP techniques to binary code search can automatically generate semantic embeddings of instruction sequences. While we are not the first to apply NLP into binary code search [10], [35], [60], we are the first to refine instruction sequences to preserve main semantics in static BCSD approaches. For the characteristics of binary functions, we customize the language model and propose two pre-training tasks based on binary code characteristics, which makes our model more robust. (4) The previous work [34] proved that CFG-based GNNs are effective ways of solving the binary function similarity problem. To generate function embeddings, we utilize GCNs to aggregate basic block semantics and integrate all basic block embeddings based on CFGs, which makes generated function embeddings include CFG structure features and basic block semantics.

In Section IV-H, we find outdated libraries used in firmware images. Updating these outdated libraries directly to the latest versions may break file dependencies in firmware. VulHawk can detect these known vulnerable functions. Furthermore, we can generate hot patches at the binary level to prevent these known vulnerabilities. We leave this as future work.

VI. Related Work

In this section, we briefly survey additional related work. We focus on approaches using code similarity for code search and known vulnerability discovery without source code.

Mono-architecture Approaches. Binary code search for mono-architecture binaries has achieved a lot. TEDEM [43] introduces tree edit distances to measure code similarity at the level of basic blocks. BinHunt [15], and CoP [32] compare binary code similarity using symbolic execution and a theorem prover. But these approaches are computationally expensive and thus not applicable to large function repositories. Tracelet [7] decomposes binary functions into continuous traces and uses the edit distances between two traces to measure their similarity. Other methods [20], [23], [37], [38] use statistical features, syntax features, and structure features to match similar binary functions. Inspired by NLP techniques, many researchers [10], [11], [27] introduce language models to extract the semantics of binary code. DeepBinDiff [11] considers assembly instructions as words and uses Word2Vec CBOW model [36] to generate semantic embeddings for binary code. Asm2Vec considers functions as documents, and tokens (opcodes or operands) as words in the document, and utilizes a Distributed Memory Model of Paragraph Vectors (PV-DM) model [26] to generate function embeddings. They represent binary functions as high-dimensional numerical vectors, facilitating the search for similar candidate functions in large function repositories. However, for IoT firmware images from different architectures, binary code search methods are required to support finding similar functions across architectures.

Cross-architecture Approaches. Recently, researchers start to address the challenge of cross-architecture binary code search. Traditional approaches select architecture-robust features from statistical features, syntax features, and structure features to calculate the similarity of binary code. BinDiff [61], as a state-of-the-art commercial binary code similarity detection tool, extracts the number of basic blocks, string references, and the structure of call graphs to calculate the similarity between binary functions. DiscovRE [12] utilizes CFG-based matching to find similar functions, but graph matching brings expensive computation. Esh [5] uses SMT solver based on data-flow slices of basic blocks to verify function similarity, which does not support large function repositories. Genius [14], introducing machine learning, considers statistical features as attributes of CFGs to graph embeddings for binary code search. Gemini [55] improves Genius by adopting neural networks to generate function embeddings to match similar functions. Gemini and Genius both use hand-crafted features which require rich experience and expert knowledge to match similar functions. InnerEye [60] treats binaries from different architectures as different languages and uses neural machine translation based Word2Vec model [36] to calculate binary code similarity across specified architectures (x86 and arm). SAFE [35] trains its language model using binaries from multiple architectures to search binary code across architectures. However, SAFE has heavy OOV issues in practice, so it gets poor performance in cross-architecture tasks. Trex [41] proposes micro-traces including instructions and dynamic values, and learns function execution semantics from their micro-traces for cross-architecture similar function matching.

According to different usage requirements, IoT firmware images from different architectures are generated by different compilers with different optimization levels. There are many differences between these binaries caused by different compilers and optimization levels. The existing methods using a fixed trained model may work for specific optimization levels and compilers, but they are still difficult to achieve good performance against various compilers and optimization levels.

VII. Conclusions

In this paper, we propose a cross-architecture binary code search approach VulHawk. It contains an intermediate representation function model based on RoBERTa and GCNs for generating function embeddings. For robustness against different file environments, we propose a divide-and-conquer strategy and introduce the entropy from the information-theoretic perspective to identify the file environments of binary code. We propose an entropy-based adapter to transfer function embeddings into the same intermediate file environment to alleviate the differences caused by different compilers, optimizations, and architectures. In the progressive search strategy, the similarity calibration supplements vulnerability detection using fine-grained level features to reduce false positives caused by patched functions. Extensive evaluations demonstrate that VulHawk outperforms state-of-the-art approaches in seven tasks. The divide-and-conquer strategy effectively improves the robustness of VulHawk against compilers, optimization levels, and architectures. The 1-day vulnerability detection experiment shows VulHawk's high performance in detecting vulnerabilities.

Acknowledgment

We thank the anonymous reviewers for the helpful comments. This work is partially supported by the National University of Defense Technology Research Project (ZK20-17, ZK20-09), the National Natural Science Foundation China (62272472, 61902405), the HUNAN Province Natural Science Foundation (2021JJ40692), the National Key Research and Development Program of China under Grant No. 2021YFB0300101, and the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013).

References

[1] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, "Simgnn: A neural network approach to fast graph similarity computation," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, 2019, pp. 384–392.

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[3] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, vol. 6, 1993.

[4] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware variant detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 307–317, 2013.

[5] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 266–280.

[6] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM New York, NY, USA, 2018, pp. 392–404.

[7] Y. David and E. Yahav, "Tracelet-based code search in executables," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 349–360, 2014.

[8] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, vol. 1, no. Mlm, pp. 4171–4186, 2019.

[9] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. New York, New York, USA: ACM Press, 2016, pp. 461–470.

[10] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[11] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing," in *Proceedings of the 27rd Symposium on Network and Distributed System Security (NDSS)*, 2020.

[12] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proceedings 2016 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2016, pp. 21–24.

[13] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 2014, pp. 78–87.

[14] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-based Bug Search for Firmware Images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. New York, New York, USA: ACM Press, 2016, pp. 480–491.

[15] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[16] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 896–899, 2018.

[17] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[18] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 611–620.

[19] H. Huang, A. M. Youssef, and M. Debbabi, "Binsequence: fast, accurate and scalable binary code reuse detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 155–166.

[20] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 81–96.

[21] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[22] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 342–352.

[23] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," *IEEE International Working Conference on Mining Software Repositories*, pp. 329–338, 2013.

[24] T. N. Kipf and M. 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*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[25] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[26] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[27] X. Li, Q. Yu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[28] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[29] J. Lin, D. Wang, R. Chang, L. Wu, Y. Zhou, and K. Ren, "Enbindiff: Identifying data-only patches for binaries," *IEEE Transactions on Dependable and Secure Computing*, no. 01, pp. 1–1, 2021.

[30] B. Liu, W. Li, W. Huo, F. Li, W. Zou, C. Zhang, and A. Piao, "αDiff: Cross-version binary code similarity detection with DNN," *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 667–678, 2018.

[31] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

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

[33] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[34] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem."

[35] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[36] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[37] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 253–270.

[38] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing*, 2005, pp. 314–318.

[39] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[40] T. of Bits. (2021) Github - McSema. [Online]. Available: https://github.com/lifting-bits/mcsema

[41] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[42] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?" *arXiv preprint arXiv:2105.04297*, 2021.

[43] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 406–415.

[44] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multiple architectures," *IEEE Access*, vol. 9, pp. 163 461–163 475, 2021.

[45] H. Rays, "IDA Pro," https://www.hex-rays.com/products/ida/, May 2021.

[46] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: http://arxiv.org/abs/1908.10084

[47] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[48] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 114–138.

[49] Synopsys, "Open source security and risk analysis report," https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf, 2022.

[50] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.

[51] P. Team, "Github - PyTorch Geometric," https://github.com/pyg-team/pytorch_geometric, 2022.

[52] Transformers, "Github - Transformers," https://github.com/huggingface/transformers, 2021.

[53] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 1–13.

[54] H. Xiao, "bert-as-service," https://bert-as-service.readthedocs.io, 2022.

[55] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. Dallas,TX, USA: ACM Press, 2017, pp. 363–376.

[56] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 462–472.

[57] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, "Codee: A tensor embedding scheme for binary code search," *IEEE Transactions on Software Engineering*, 2021.

[58] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 224–236.

[59] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, pp. 1145–1152, 2020.

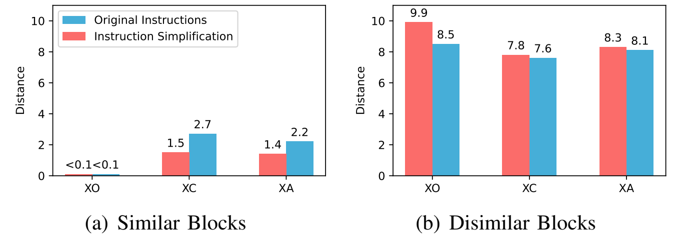
[60] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs," in *Proceedings 2019 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2019.

[61] Zynamics, "Bindiff," https://www.zynamics.com/bindiff.html, 2021.

APPENDIX

A. Ablation Study on the Instruction Simplification

Fig. 14 shows the results of the ablation study on instruction simplification. Red bars indicate the distances between the basic block embeddings with the instruction simplification, and blue bars indicate the distances between the original basic block embeddings. We carefully use source code lines of DWARF (debugging with attributed record formats) information to label basic block similarity. We randomly pick 10,000 similar block pairs and dissimilar block pairs from the XO, XC, and XA testing datasets, respectively. With the help of instruction simplification, the distances between similar blocks are brought closer (Fig. 14(a)), and dissimilar blocks are drawn farther apart (Fig. 14(b)). For example, the average distance between similar blocks reduces by 44.4%, and the average distance between dissimilar blocks increases by 2.6% in the XC task. This is because the instruction simplification refines microcode sequences and increases the proportion of main semantics, which improves our language model's performance on basic block similarity detection.



(a) Similar Blocks (b) Disimilar Blocks

Fig. 14. Results of ablation study on the Instruction simplification. The blue bars represent distances between the original basic blocks, and the red bars represent distances between the blocks with instruction simplification.

B. Vulnerability Details

We detail the vulnerabilities which are detected and checked in the 1-day vulnerability detection.

1) 1-day Vulnerabilities: In Table VII, we list the firmware and vendors to which these vulnerabilities belong. We find that adjacent models of the same vendor usually have similar vulnerable functions. For example, the adjacent models of NetGear R6400, R6900, R7000, R7000P, and R8000 all contain vulnerabilities CVE-2016-2182, CVE-2016-6303, CVE-2016-8618, CVE-2016-8622, CVE-2018-1000301, and CVE-2021-23840. We carefully analyzed the reason and found that these firmware images use similar compilation chains (GCC-3.5) and library dependencies, which easily package the same vulnerable functions into the generated firmware images. We have safely reported the discovered vulnerabilities in the experiments and their patches to the corresponding vendors to help fix them.

2) New Vulnerabilities: We use VulHawk to perform vulnerability detection in the latest projects and find a suspected stack overflow in `OpenSSL-3.1.0/ssl/ssl_lib.c`. Fig 15 shows the vulnerability detail. The return type of `strlen` function is `size_of`, while the type of `n` is `int`. When the return value is over the upper bound of an integer (2^{31} −

```
1  char *SSL_get_shared_ciphers(const SSL *s, char *buf, int size){
2      ...
3      int n;
4      c = sk_SSL_CIPHER_value(clntsk, i);
5      n = strlen(c−>name); // Forced type conversion
6      if (n + 1 > size) {
7          ...
8          return buf;
9      }
10     strcpy(p, c−>name); // Overflow!
11     ...
12 }
```

Fig. 15. A suspected stack overflow vulnerability in OpenSSL-3.1.0.

TABLE VII. Details of 1-day vulnerability detection.

#	CVE	Vulnerable function	Confirmed #	Affected firmware
1	CVE-2015-1789	X509_cmp_time	3	TP-Link: TL-WAR458, TL-WVR300, TL-R473;
2	CVE-2015-0286	ASN1_TYPE_cmp	3	TP-Link: TL-WAR458, TL-WVR300, TL-R473;
3	CVE-2016-0797	BN_dec2bn	4	TP-Link: Deco M4, TL-WAR458, TL-WVR300, TL-R473;
	CVE-2016-0797	BN_hex2bn	4	TP-Link: Deco M4, TL-WAR458, TL-WVR300, TL-R473;
4	CVE-2016-2176	X509_NAME_oneline	4	TP-Link: Deco M4, TL-WAR458, TL-WVR300, TL-R473;
5	CVE-2016-0798	SRP_VBASE_get_by_user	4	TP-Link: Deco M4, TL-WAR458, TL-WVR300,TL-R473;
6	CVE-2016-2182	BN_bn2dec	14	D-Link: DIR-842; NetGear: R7000, R7000P, XR300, EX7000, R8000, R6400, R6900, R8500, R6700; TP-Link: Deco M4, TL-WAR458, TL-WVR300, TL-R473
7	CVE-2021-23840	EVP_DecryptUpdate	17	D-Link: DIR-842; NetGear: X7300, RBR50, RBS40, RBS20, RBR20, R7000, R7000P, XR300, EX7000, R8000, R6400, R6900, R8500, R6700, EX6150; TP-Link: Deco M4, TL-WAR458, TL-WVR300, TL-R473;
8	CVE-2016-6303	MDC2_Update	10	D-Link: DIR-842; NetGear: R7000, R7000P, XR300, EX7000, R8000, R6400, R6900 R8500, R6700;
9	CVE-2016-8622	curl_easy_unescape	10	NetGear: R7000, R7000P, XR300, EX7000, R8000, R6400, R6900, R8500, R6700; TP-Link: Deco M4
10	CVE-2018-1000301	Curl_http_readwrite_headers	9	NetGear: R7000, R7000P, XR300, EX7000, R8000, R6400, R6900, R8500, R6700;
11	CVE-2016-8618	alloc_addbyter	10	D-Link: DIR-842; NetGear: R7000, R7000P, XR300, EX7000, R8000, R6400, R6900, R8500, R6700;
12	CVE-2021-22924	create_conn	1	D-Link: DIR-842;

1), `n` may be negative because of an integer overflow. The expression `n+1>size` is not satisfied, and the execution of `strcpy` will result in a stack overflow. The `SSL_get_shared_ciphers` function is an exported function in `libssl.so` of OpenSSL-3.1.0 project. When a code calls this function directly, it represents a security issue.

```
1 mov   rax, [rbp+var_40]
2 mov   rdi, [rax+8]
3 call  strlen
4 mov   [rbp+var_48], eax
5 mov   eax, [rbp+var_48]
6 add   eax, 1
7 cmp   eax, [rbp+var_1C]
8 jle   loc_7722

// Psuedocode
n = strlen(s)
if (n+1 > size){...}
strcpy()
```

O0-Clang-x86_64

```
1 mov   r13, [rbp+8]
2 mov   rdi, r13
3 call  strlen
4 mov   r15, rax
5 cmp   r14d, r15d
6 jle   loc_50C5

// Psuedocode
n = strlen(s)
if (size > n) strcpy()
...

// Psuedocode
n = strlen(s)
if (n >= size){...}
strcpy()
```

O3-Clang-x86_64

Fig. 16. The binary code of the discovered vulnerability in OpenSSL-3.1.0.

When coding this patch, we found an interesting issue. We fix the type of `n` into `size_of`, the binary file from `O3-Clang-x86-64` is fixed, while the binary file from `O0-Clang-x86-64` is still at risk of overflow. Fig. 16 shows that `Clang` optimizes `n+1>size` to `n>=size` when compiling with the `O3` option, eliminating the possibility of overflow due to `n+1` when `n` equals $2^{32}-1$. Therefore, the fixed code modifies the type of `n` to `size_of` and changes the expression `n+1>size` to `n>=size`.

Finally, we have safely reported the discovered vulnerabilities in our experiments to the corresponding vendors and developers.