PMatch: Semantic-based Patch Detection for Binary Programs

Zhe Lang*†, Shouguo Yang*†‡, Yiran Cheng*†, Xiaoling Zhang*†, Zhiqiang Shi*†‡, Limin Sun*†
*Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
†School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
{langzhe, yangshouguo, chengyiran, zhangxiaoling, shizhiqiang, sunlimin}@iie.ac.cn, ‡corresponding author

2021 IEEE International Performance, Computing, and Communications Conference (IPCCC) | 978-1-6654-4331-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/IPCCC51483.2021.9679443

***Abstract*—Binary function matching has been proposed to detect the known vulnerabilities. However, the high similarity between the vulnerable and patched versions leads to a large of false positives. Patch detection is proposed to improve the accuracy of function matching by identifying the patched functions from matching results. However, the accuracy of existing methods decreases significantly due to the function changes introduced by high compiler optimization levels.**

In this paper, we propose PMatch, a method based on code semantic similarity to detect the patched binary functions. Firstly, PMatch extracts patch-affected code snippets from the patched binary function. Secondly, PMatch leverages a novel unsupervised sentence embedding technique in Natural Language Processing (NLP) to generate the semantic representations of binary code. Finally, PMatch matches the patch-affected code snippets with target blocks obtained by function diffing. To evaluate PMatch, we collect 101 CVEs and compile 304 binary programs with 4 different optimization levels. PMatch achieves an 86.43% average accuracy in detecting the patched functions, which outperforms the state-of-the-art work, and costs only 65.14ms per function. Besides, at the O3 high optimization level, PMatch achieves an accuracy improvement of over 20%.

***Index Terms*—Binary Function Matching, Patched Detection, Code Semantic Similarity, Natural Language Processing**

I. Introduction

Software vulnerabilities are one of the largest threats to personal security and even national security today, e.g., the heartbleed vulnerability [1]. The number of vulnerabilities submitted to the Common Vulnerabilities and Exposures (CVE) database is increasing every year e.g., 6,457 in 2016 and almost triple to 18,377 in 2020 [2]. Due to the reuse of open-source software [3], vulnerabilities spread among different projects, firmware, and systems widely [4]. To detect such vulnerabilities, binary function matching techniques are proposed to retrieve vulnerable functions [5]–[11]. However, the function matching results contain a large of false positives due to the high similarity between the vulnerable and patched functions [12]. Patch detection is proposed to reduce false positives of function matching by identifying the patched functions from matching results. However, it is difficult to distinguish the patched functions from vulnerable functions, since patches usually introduce small code changes to patched functions [13].

Recently, a series of approaches based on binary program analysis have been proposed for patch detection. SPAIN [14] identifies patched blocks from all changed blocks between patched functions and vulnerable functions. BinXray [12] extracts the patch semantic signatures from the changed blocks induced by patches to detect the patched functions. However, the accuracy of these methods decreases significantly due to the substantial function changes introduced by high compiler optimization levels. Specifically, high optimization levels introduce binary changes which are irrelevant to patches as shown in Section II-A.

Such binary changes increase the suffering of patch detection. First, the high optimization levels make the patched block localization methods by binary diffing (i.e., BinXray) inaccurate. To locate patched blocks as the ground truth, we take full advantage of the source code information. Specifically, we build the mapping relationship between source code and compiled binary code as shown in Figure 7 and name mapped binary code as patch-affected code snippets.

Second, high optimization levels introduce lots of noises (i.e., the binary changes unrelated to patches), which makes it difficult to detect patches. However, the patched blocks within the target function usually contain similar semantics with the patch-affected code snippets. To mitigate the effects of the noises, we leverage semantic similarity to detect the patched blocks in the target function. Then we turn our attention to the natural language processing (NLP) techniques. Inspired by works of InnerEye [10] and Asm2Vec [15], we treat instructions as words and code blocks as sentences. We apply the unsupervised sentence embedding technique in NLP to learn the semantic representations of code blocks. Our unsupervised approach is more scalable than the supervised methods based on neural networks since it does not require labeling data and training the model.

In this paper, we propose PMatch, a method based on code semantic similarity to detect the patched binary functions. We first construct a code database that contains lots of patch-affected code snippets. We utilize the code snippets from the database to match suspicious patch blocks. Note that the patch-affected code snippets are treated as normal code blocks during the detection. Then PMatch leverages the CBOW neural network model [16] to learn the embeddings of instructions in blocks. Further, PMatch applies smooth inverse frequency (SIF) [17] to generate the semantic feature vectors of code

blocks (snippets) based on instruction embeddings. We find that the unsupervised approach is more accurate and efficient than the state-of-the-art deep learning-based supervised method INNEREYE in the evaluation. Finally, PMatch matches the patch-affected code snippets with target blocks by measuring their semantic similarities between their feature vectors to detect patched functions. PMatch regards target functions as patched functions if they contain patched blocks.

We summarize our contributions as follows:

- We apply a novel unsupervised sentence embedding approach SIF to generate the semantic representations of code blocks. We demonstrate that our unsupervised approach is more accurate and efficient than the state-of-the-art supervised method by conducting a comparative test.
- We implement the prototype of PMatch for patch detection. We collect 101 CVEs of OpenSSL and FFmpeg projects and build a dataset for evaluation. The dataset consists of 304 binary programs complied with 4 different optimization levels and 5,925 functions to be tested. PMatch achieves an 86.43% average accuracy in detecting the patched functions, which outperforms the state-of-the-art work BINXRAY, and costs only 65.14ms per function. Besides, at the O3 optimization level, PMatch achieves an accuracy improvement of over 20% compared to BINXRAY. We have open sourced our method [18].

II. PRELIMINARY

A. A Binary Diffing Case

We use CVE-2014-0195 [19] in OpenSSL as a binary diffing case. The patch of CVE-2014-0195 introduces about 2 source code changes to the $dtls1\_reassemble\_fragment$ function. Then Table I shows the binary diffing results obtained by Diaphora [20]. In particular, the number of changed blocks increases substantially from 3 to 19 at the O3 high optimization level compared to the O0 without optimization level. As Figure 1 shows, high compiler optimization levels can result in changes to the binaries of unchanged source code (line 8) within patch functions. But the binaries of the source code (line 8) do not have any difference between both versions at the O0 optimization level. Therefore, at high optimization levels, the number of changed blocks obtained by function diffing increases and most of them are irrelevant to the patches.

TABLE I: The number of changed blocks at different optimization levels (O0 and O3). V_BBlocks denotes **V**ulnerable function **B**asic **Blocks** and P_BBlocks denotes **P**atched function **B**asic **Blocks**, and C_Blocks denotes **C**hanged **Blocks** at different optimization levels.

Opt_Level	# of V_BBlocks	# of P_BBlocks	# of C_Blocks
-O0	56	57	3
-O3	56	56	19

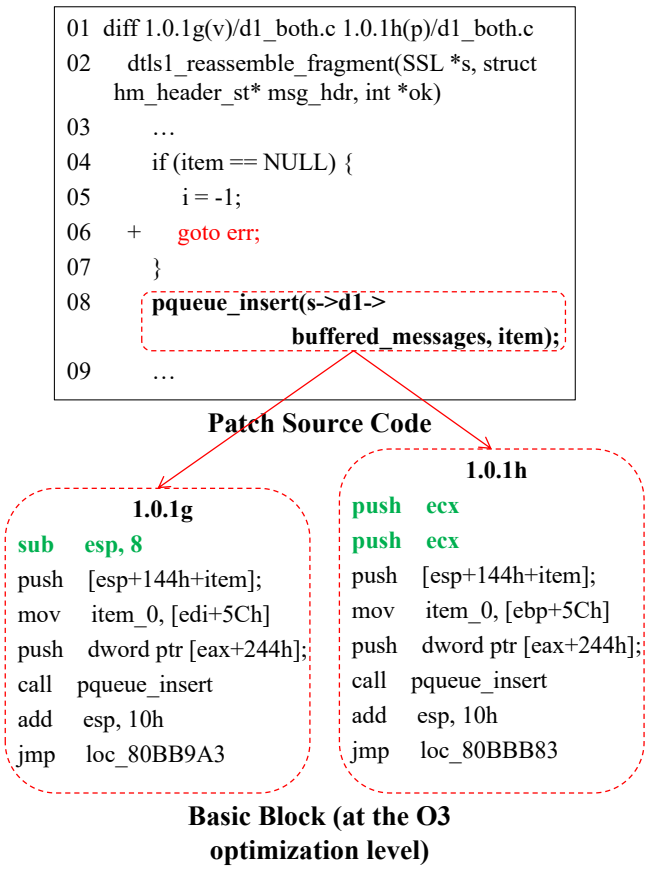


Fig. 1: A case of a binary change that is irrelevant to the patch. After the patch (of CVE-2014-0195) is introduced to the vulnerable function $dtls1\_reassemble\_fragment$, the binary of the unchanged source code within the function may be changed due to high compiler optimization levels. The red code denotes patches in the source code and the green code denotes different assembly instructions.

B. Patch-Affected Code Snippet

We use CVE-2016-2182 [21] and CVE-2015-3197 [22] as the cases of patch-affected code snippets as Figure 7 shows. Figure 7a provides a case of the mapped binary code from addition source code (line 4-5) as the patch-affected code snippets. In this case, the code snippets are patched blocks. Figure 7b provides a case of the mapped binary code from modification source code (line 3) as the patch-affected code snippets. In this case, the code snippets are not identical to patched blocks in syntax.

C. Overview

An overview of PMatch is presented in Figure 2, which consists of three main components. Taking the target and vulnerable binary functions as input, PMatch first locates candidate different blocks by utilizing the program diffing tool. Then PMatch leverages an unsupervised sentence embedding approach to generate semantic feature vectors of patch-affected code snippets and candidate different blocks respectively.

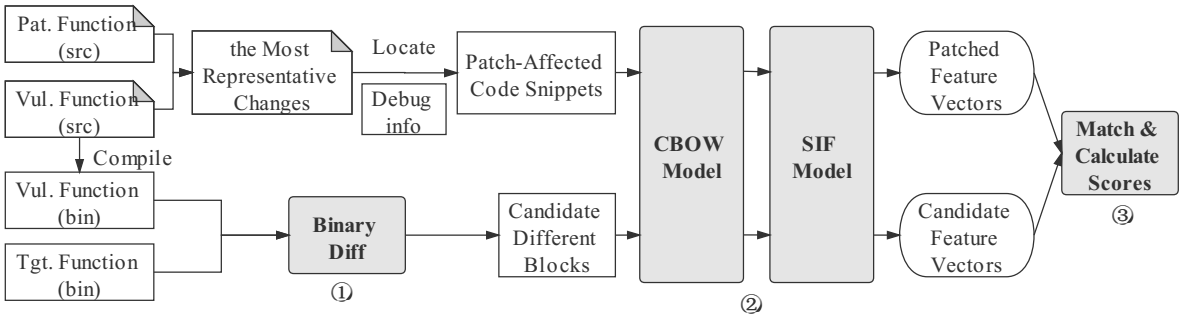


Fig. 2: The workflow of PMatch is composed of three main components: diffing binary functions (①), generating feature vectors (②), and calculating semantic similarity (③). The src denotes source code, and bin denotes binary.

Finally, PMatch determines whether the target function has been patched through the semantic similarity between the patch-affected code snippets and candidate different blocks.

1) Diffing Binary Functions: Given the target function and the corresponding vulnerable functions, this component utilizes the disassembler tool IDA PRO [23] and the binary program diffing tool Diaphora [20] for preliminary filtering to obtain multiple changes named candidate different blocks within the target function.

2) Generating Feature Vectors: This component leverages the NLP techniques to generate the semantic representations of code blocks. Note that we treat patch-affected code snippets as normal code blocks. In detail, the component first uses the CBOW neural network model [16] to learn the embeddings of assembly instructions in blocks. Then the component applies SIF [17], an unsupervised sentence embedding approach, to generate semantic feature vectors of blocks and snippets based on instruction embeddings. These feature vectors are used to calculate similarity in the next step.

3) Calculating Semantic Similarity: This component first matches the patch-affected code snippets with candidate blocks and calculates similarity scores through the cosine distance between their feature vectors. Further, the component determines whether the target function is patched or un-patched by computing the weighted average score between patched feature vectors and candidate feature vectors.

III. Methodology

A. Diffing Binary Functions

PMatch diffs the target and the corresponding vulnerable binary functions by utilizing the program differing tool Diaphora [20] to obtain multiple candidate different blocks. The vulnerable and patched code database first provides the last vulnerable version corresponding to the target function in Section III-A. Then this code database offers the patch-affected code snippets in Section III-B. In this section, we first introduce the details of building this code database.

We build a code database, which collects multiple CVE [24] vulnerable functions and patch-affected code snippets. The real-world CVE vulnerability descriptions are mostly in the form of source codes. To locate the code snippets in the patched functions, we leverage debug information to map source code lines to binary instructions. In detail, we first proceed to diff the source codes of the vulnerable and patched functions. Further, we pick out the most representative changed source lines and use debug information to obtain their binary instructions as the patch-affected code snippets. In particular, we regard input checks as the representative changes since input sanitization checks are usually the most common security patch patterns [25]. Then we leverage the symbol table to locate the vulnerable functions. Finally, we disassemble the patch-affected code snippets and vulnerable functions and stored them in the code database.

To obtain candidate different blocks, we first disassemble the target binary function by utilizing IDA PRO [23]. Then we take the last version of the corresponding vulnerable function from the code database. There are usually some same basic blocks between these two functions. So we first compute the hash values of each basic block and remove the blocks sharing the same hash value in the target and vulnerable functions. Diaphora is an advanced IDA PRO plugin to diff binary programs. Then we utilize the plugin to implement a module to diff the rest of the target and vulnerable functions. Thus we obtain multiple differences within the target function as candidate different blocks.

B. Generating Instruction Embedding

PMatch uses the word2vec technique to learn the word embedding. Further, the word embedding is a distributed digital representation, which captures semantic features of a range of contexts to represent a word. The CBOW [16] is a word2vec model based on neural networks to learn word embeddings. In particular, we apply this model to generate semantic embeddings of assembly instructions.

The continuous bag-of-words (CBOW) neural network model has been proposed to learn word embeddings through unsupervised learning. The CBOW model in PMatch is mainly composed of an embedding layer and a softmax layer as shown in Figure 3. Moreover, the dotted lines in the figure denote that the loss is propagated backward to update the weights of the embedding and softmax layers. Given a sliding window consisted of surrounding words *w(t-2)*, *w(t-1)*, *w(t+1)*, *w(t+2)*, the objective of the CBOW model is learning the word representation to predict the center word *w(t)* within the

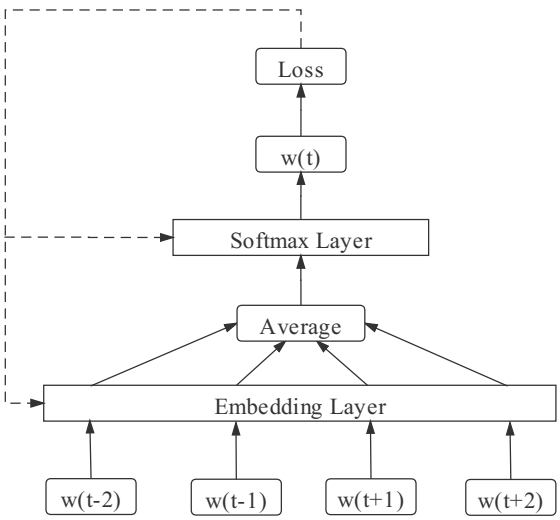


Fig. 3: The CBOW model.

window. The model first slides the contextual window in the input word sequence to obtain the contexts as input and the current word as output. Meanwhile, each word is represented as the one-hot encoding of size V, which only exists one non-zero element 1. Then this model queries the semantic embeddings of input words from the embedding layer, which is a word embedding matrix denoted by $W \in R^{V \times D}$, where V is the number of vocabularies in the table and D is the dimension of the word embedding. Further, the model averages the embeddings and uses the softmax function to obtain the predicted output that also is the one-hot vector of size V. Finally, training on a sequence of T words, the purpose of the CBOW model is to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T} log\ p(w_t|c_t) \tag{1}$$

where w_t is the center word and c_t is the context set of w_t. The log probability is defined by softmax function:

$$p(w_t|c_t) = \frac{exp(logit(x_n)_{w_t})}{\sum_{w_k\ \in\ vocabulary} exp(logit(x_n)_{w_k})} \tag{2}$$

where $logit(x_n)_{w_t}$ and $logit(x_n)_{w_k}$ are the scores for non-zero index in one-hot vectors of w_t and w_k respectively.

During the process of applying the CBOW model to generate embeddings of instructions, we encounter some problems. Firstly, an instruction contains zero or more operands that are generally expressed as constants, strings, registers, memory addresses, or other symbols. However, the complexity of kinds of operands dramatically increases the possibility of the out-of-vocabulary (OOV) case. To address the problem, we are inspired by INNEREYE [10] to preprocess the instructions by using several different symbols to replace the corresponding operands. Additionally, we also preprocess the memory address and memory data operands with the symbols $\langle ADR \rangle$ and $\langle MEM \rangle$ respectively to remain more semantics in the preprocessed instructions.

Secondly, the neural network model will constantly update the weights of neurons according to the loss of each epoch, to improve the accuracy of prediction. It is the size of vocabulary that usually determines the scale of the weight matrix. Thus, the large number of instructions in vocabulary causes an expensive cost of maximizing log probability objective during training the neural network. To overcome the problem, we leverage negative sampling based on unigram distribution [26], [27]. In detail, the negative sampling mechanism randomly regards a small part of instructions as negative words and only updates the neural weights related to the target and negative words each time, which significantly reduces the time costing in the training process. In addition, the unigram distribution makes the high-frequency instructions more likely to be regarded as negative words.

In this section, we transform the assembly instructions of code blocks into instruction embeddings by leveraging the word2vec model CBOW.

C. Generating Block Embedding

PMatch leverages the sentence embedding approach to complete a sent2vec task. In particular, we use SIF [17] to learn embeddings of code blocks, which is an unsupervised sentence embedding approach in NLP. The block embeddings are feature vectors that represent semantic of the code blocks. Thus, the embeddings of two code blocks with similar semantics are close to each other in high-dimensional space.

The smooth inverse frequency (SIF) is a novel sentence embedding approach that is different from neural network methods. Moreover, SIF is a simple but formidable approach to learn sentence representations. In detail, SIF represents the sentence simply by the weighted average result of word embeddings, where the result is modified by using PCA/SVD. Although the approach is unsupervised, it beats many sophisticated supervised methods including a series of RNN models in sentence similarity tasks.

SIF proves that maximum a posteriori (MAP) estimate of the unknown parameter sentence embedding is approximately the weighted averages of embeddings of the words in the sentence. In particular, the weight is defined as:

$$weight = \frac{a}{p(w) + a} \tag{3}$$

where a is a scalar hyperparameter and $p(w)$ is frequency of the word in a corpus. Then SIF computes the first principal component of the embedding matrix consisted of a set of sentences. Further, it subtracts the projection of the sentence on the first principal component from its embedding to obtain the final embedding. It will remove the information of high-frequency words that are often related to syntax but have less relevant semantic.

We regard a code block as a sentence and leverage SIF to generate the embeddings of code blocks. To improve the efficiency of embedding generation, we count the occurrence

frequency of each assembly instruction in our corpus (Dataset I) and calculate the weight of each instruction by Equation 3 in advance. Then we generate a block embedding by computing the weighted average result of instruction embeddings, where the instructions are from the code block. This block embedding is a semantic feature vector of the code block since the embedding is the optimal combination of their instruction semantic vectors to represent semantic of the block. Finally, we subtract the projection of the block on the first principal component of the embedding matrix from its embedding, to obtain the final semantic representation of the code block.

In this section, we apply an unsupervised sentence embedding approach to obtain the semantic feature of the code block.

D. Calculating Semantic Similarity

PMatch first matches each patch-affected code snippet with candidate different blocks. Then PMatch determines whether the target function has been patched through the semantic similarity of the matching pairs.

In detail, we first search for the most similar block to the patch-affected code snippet in the candidate different block set, by the cosine distance between their semantic feature vectors. Note that we treat the patch-affected code snippets as normal code blocks. If a matching pair of similar blocks exists, two blocks in this pair will be removed from the patch-affected code snippet set and the candidate block set respectively. If no matching pair exists, only the patch-affected code snippet will be removed from the patch-affected code snippet set. This matching process does not end until the patch-affected code snippet set has been cleared. Then we calculate the semantic similarity through the weighted average score that is defined as:

$$\sum_{s \in ps} w_s \cdot score_s \tag{4}$$

where the weight w_s is the ratio of the number of instructions in the current patch-affected code snippet s to the total number of instructions in the patch-affected code snippet set ps, and the $score_s$ is the cosine distance between the feature vectors of the current patch-affected code snippet s and its matched block. Finally, if the semantic similarity score exceeds the threshold, we predict the target function to be a patched function.

IV. Evaluation

A. Experimental Settings

We utilize IDA PRO 7.5 [23] and its plug-in Diaphora 2.0 [20] to disassemble binary programs and diff two assembly functions. Like the work FIBER [28], we require to locate the binary patch-affected code snippets as the ground truth. In particular, we use an address conversion tool addr2line 2.30 to complete this task by debug information. We implement a patch detection prototype PMatch in Python 3.6.7 with TensorFlow 1.13.1 [29] and sklearn 0.20.3 [30]. TensorFlow provides plenty of neural network APIs and sklearn offers lots of machine learning-based algorithm models. Our experiments are performed on a server running the Ubuntu 16.04.12 operating system with two Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz, 256 GB RAM and 6 GeForce GTX 1080 Ti GPUs.

In the experiments, we aim to answer the following research questions:

RQ1: Why Patch applies an unsupervised sentence embedding approach to encode blocks?

RQ2: How accurate is PMatch compared to the state-of-the-art work for patch detection?

RQ3: How accurate is PMatch in the cross-optimization level experiment?

RQ4: What is the performance of PMatch in patch detection?

B. Dataset

1) Dataset I: The dataset is used to train and evaluate the word2vec models. It consists of numerous assembly instruction blocks. In particular, we compile the binary OpenSSL (v1.0.1u) and Binutils (v2.34) software from their source code by using x86-32 GCC (v5.4.0) with optimization levels O0, O1, O2, and O3. In total, we obtain 70,628 code blocks and 820,189 assembly instructions.

2) Dataset II: The dataset is used to evaluate the accuracy of INNEREYE [10] and our prototype PMatch. It is composed of multiple target binary functions to be detected. In detail, we compile 84 OpenSSL binaries from version 1.0.1a to 1.0.1u and 220 FFmpeg binaries from version 3.0.1 to 3.4.8 by using x86-32 GCC with four different optimization levels (i.e., O0, O1, O2, and O3). Meanwhile, we collect 101 CVE vulnerability information from OpenSSL and FFmpeg. Further, we retrieve the name of the vulnerable function from every CVE vulnerability description. Then we select all versions of the vulnerable function except the last vulnerable version as its target functions, which guarantees the adequacy of the detection of target functions. However, functions are more likely to be inlined at higher optimization levels. We remove the inline functions from these target functions and obtain a total of 5,925 target functions compiled with four different optimization levels.

C. Baseline Methods

We choose INNEREYE and BINXRAY as baseline methods to compare with PMatch.

INNEREYE is the state-of-the-art deep learning-based supervised method for code inclusion detection. If the query code snippets are the patched blocks, code inclusion detection is essentially the same as patch detection. So we compare our unsupervised approach SIF against the deep learning-based supervised method. INNEREYE applies a Siamese architecture [31] with employing identical LSTMs [32] to learn the semantic representations of code blocks. We re-implement the neural network model of INNEREYE with the hyper-parameters of its paper. We build the training dataset of this model from the binary code blocks on Dataset I. In particular, we obtain homologous code blocks from the binaries compiled

with different optimization levels through debug information and label them as similar pairs. Simultaneously, we label two different code blocks as dissimilar pairs. Then we convert the code blocks into embedding sequences through instruction embeddings provided by CBOW and train the model on this dataset. We redesign the calculating similarity strategy of INNEREYE so that it can accept our patched blocks as inputs. The INNEREYE's similar path exploration method is not efficient for patch detection and requires that each affected-patch snippet for the query must be a complete basic block.

BINXRAY is the state-of-the-art work for patch detection. Unlike PDIFF [33], which aims to detect kernel patches, BINXRAY focuses on the patch detection of application software. We, therefore, compare PMatch with BINXRAY in patch detection. To guarantee the fairness of this comparison experiment, we extract the relevant vulnerable and patched binary functions from OpenSSL and FFmpeg programs in Dataset II and generate the semantic signatures from such functions as the ground truth of BINXRAY.

D. Evaluation on PMatch

1) Evaluation on Instruction Embedding: The CBOW is a neural network-based model for learning the embeddings of words. Dataset I is a corpus containing over eight hundred thousand x86 assembly instructions. We train the model and evaluate the training loss of it on Dataset I. As Figure 4 shows, the loss value of the model almost stays stable after four million steps of training. Therefore, we conclude that the model has converged.

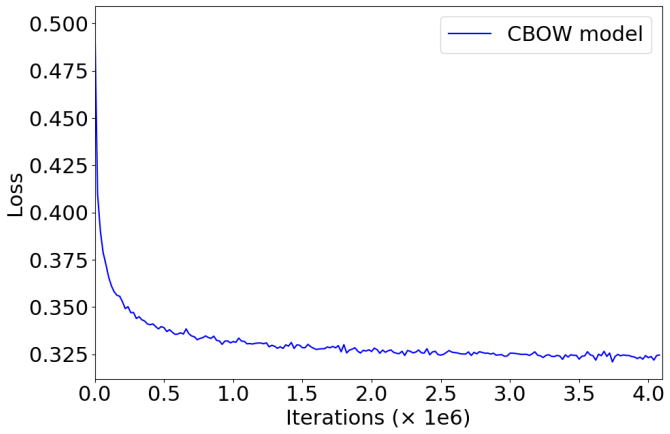


Fig. 4: The loss of CBOW model on dataset I.

We evaluate the quality of the instruction embeddings by using the t-SNE [34]. The t-distributed stochastic neighbor embedding (t-SNE) is a machine learning algorithm for dimensionality reduction that allows the high-dimensional data to be visualized in a low-dimensional space. As Figure 5 shows, the assembly instructions from the same type are closer to each other in two-dimensional space, which means that such instructions have similar distributed digital representations in high-dimensional space. Therefore, the CBOW model learns semantic embeddings of instructions effectively.

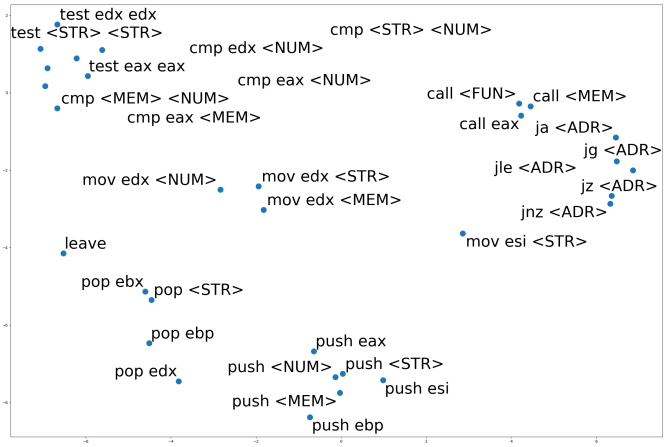


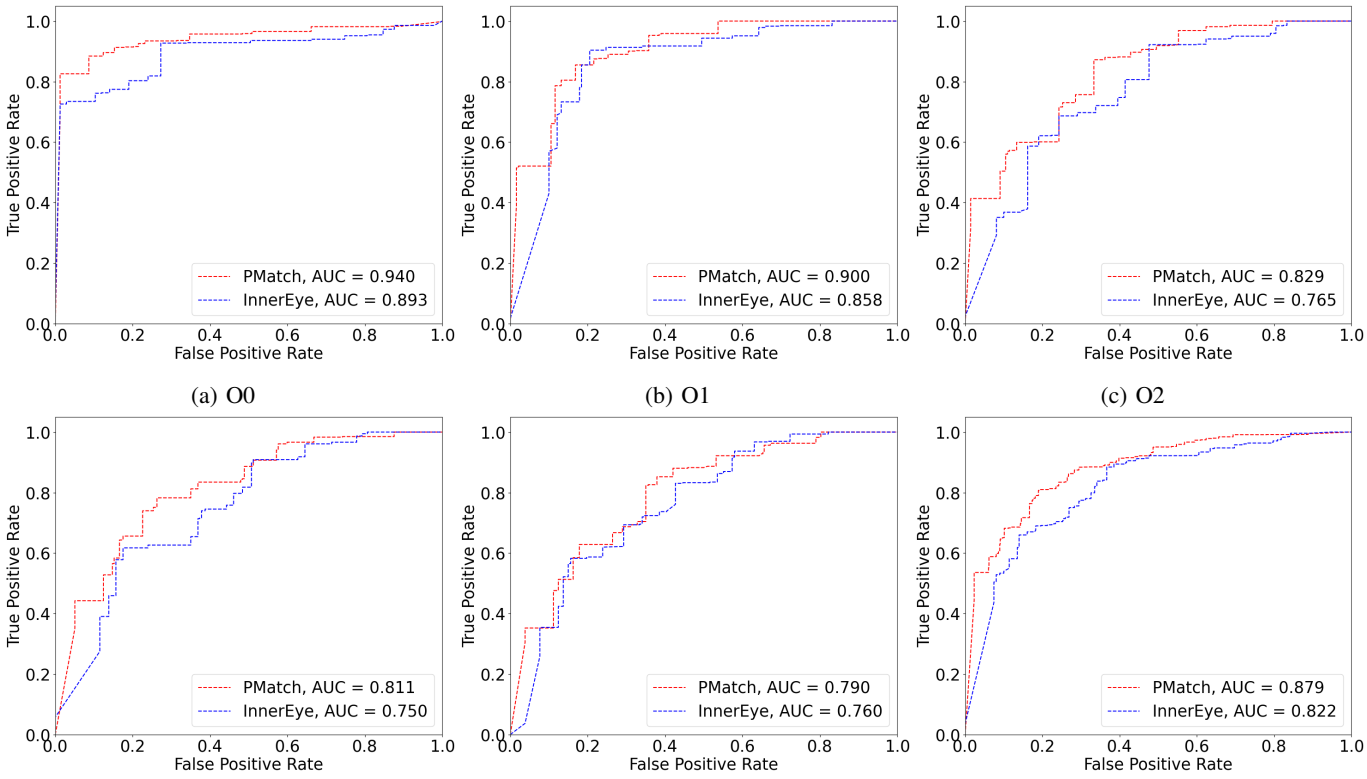
Fig. 5: The visualization of instructions in two-dimensional space.

2) Comparing with the Supervised Learning Method (RQ1): Figure 6 shows the ROC evaluation results of PMatch and INNEREYE on Dataset II. We can find that our approach achieves higher AUC values compared to the supervised learning method at four different optimization levels. Particularly, at the O2 and O3 optimization levels, the AUC values of our approach improve about 6% compared to INNEREYE. Then, we utilize the ROC curve from Figure 6f, which shows the ROC evaluation result for overall target binary functions on Dataset II, to obtain the optimal threshold (0.6) as the threshold for PMatch to determine whether the target function has been patched. Meanwhile, our approach takes less time to analyze a target function compared to INNEREYE at each optimization level as shown in Table II. Moreover, our unsupervised approach is more scalable since it does not require labeling data and training the model.

Answering RQ1: The unsupervised sentence embedding approach SIF applied in PMatch is more accurate, efficient, and scalable than the deep learning-based supervised method INNEREYE for patch detection. The AUC values of the unsupervised approach improve about 6% compared to INNEREYE at the O2 and O3 optimization levels.

TABLE II: Performance of comparing with the supervised learning method INNEREYE and the state-of-the-art work BINXRAY. Cross opt denotes cross optimization levels and Avg denotes the average time per function at different optimization levels.

Time (ms)	O0	O1	O2	O3	Cross opt	Avg
PMatch	54.80	**53.41**	59.70	73.52	97.03	65.14
INNEREYE	61.46	63.19	102.74	132.93	180.96	101.84
BINXRAY	**54.68**	58.92	**57.00**	**63.83**	**77.55**	**61.13**



(a) O0 (b) O1 (c) O2

(d) O3 (e) Cross optimization levels (f) Overall

Fig. 6: The ROC evaluation results on Dataset II.

3) Comparing with the State-of-the-art Work (RQ2): Table III shows the results of the accuracy comparison experiment. PMatch achieves an 86.43% average accuracy in detecting the patched functions, which significantly improves over 14% comparing with BINXRAY. PMatch is more accurate than BINXRAY at each optimization level. Furthermore, PMatch achieves an accuracy improvement of over 20% compared to BINXRAY at the O3 high optimization level.

The higher optimization levels result in more binary changes between the vulnerable and patched functions. Moreover, these binary changes are irrelevant to patches. BINXRAY extracts the valid trace sets consisted of changed basic blocks as the signatures for patch detection. Substantial binary changes make BINXRAY generate inaccurate signatures and increase the number of valid traces so much that it is particularly difficult to calculate the similarity between the signatures. These two reasons significantly decrease the accuracy of BINXRAY at high optimization levels. As Table III shows, BINXRAY is less accurate at higher optimization levels. However, PMatch first takes advantage of source code information and debug information to obtain accurate mapped binary code as the ground truth. Then it leverages semantic similarity to detect the patch blocks, which maintains accuracy even with more binary changes unrelated to patches (i.e., noises) at high optimization levels. We use the $dtls1\_reassemble\_fragment$ function (of CVE-2014-0195) in Section II-A as an example. At the O3 optimization level, BINXRAY detects 33 changed basic blocks between the vulnerable and patched functions, but PMatch obtains 6 changed blocks as candidate different blocks. Such a large number of changed basic blocks causes BINXRAY to fail to correctly detect patches in the target function. However, PMatch identifies the patches of the target function using semantic similarity.

In addition, the accuracy of BINXRAY deviates from its original paper since only binary programs from a single optimization level are used for its original evaluation.

TABLE III: Accuracy of comparing with the state-of-the-art work BINXRAY. Cross opt denotes cross optimization levels and Avg denotes the average accuracy at four different optimization levels.

Accuracy (%)	O0	O1	O2	O3	Cross opt	Avg
PMacth	**90.78**	**87.90**	**82.64**	**83.18**	**80.90**	**86.43**
BINXRAY	82.36	71.49	69.30	62.79	42.33	72.24

Answering RQ2: PMatch achieves an 86.43% average accuracy in detecting the patched functions, which outperforms the state-of-the-art work BINXRAY. Furthermore, at the O3 high optimization level, PMatch achieves an accuracy improvement of over 20% compared to BINXRAY.

4) Accuracy in the Cross-optimization Level Experiment (RQ3): In the real world, the optimization levels of target functions are usually unknown. The optimization levels of target and corresponding vulnerable functions may be different. Therefore, we also design a cross-optimization level experiment to evaluate the accuracy of PMatch, INNEREYE, and BINXRAY in practice. In detail, we use OpenSSL binary programs from Dataset II for evaluation. Then we take the functions complied with the O3 level as the targets to be detected. And we use the vulnerable functions and patch-affected code snippets (or signatures) compiled with the O2 level.

As Figure 6e shows, PMatch achieves a higher AUC value than the supervised learning method INNEREYE at cross optimization levels. Table III shows that PMatch achieves an 80.90% accuracy of detecting the patched functions, which still maintains accuracy compared to BINXRAY in the cross-optimization level experiment.

Answering RQ3: PMatch is still more accurate than INNEREYE and BINXRAY in the cross-optimization level experiment.

5) Performance in Patch Detection (RQ4): Table II shows the efficiency of PMatch, INNEREYE, and BINXRAY in patch detection on Dataset II. The second to sixth columns report analysis time per function of these methods at four optimization levels and cross optimization levels. The last column computes the average analysis time per function at all optimization levels. PMatch takes only 65.14ms to analyze a target function on average, a little more than 61.13ms of BINXRAY, while means that PMatch achieves a higher accuracy without adding much extra time costing. Besides, PMatch achieves less detection time compared to INNEREYE at each optimization level.

Answering RQ4: PMatch achieves a 65.14ms average analysis time in patch detection, and BINXRAY takes average detection time 61.13ms. PMatch takes a little more overhead time to achieve significant accuracy improvement.

V. RELATED WORK

Our work attempts to detect patched functions from the results of function matching to improve the accuracy of known vulnerability detection. Hence, we discuss the related work in the area of patch detection. The work can be categorized into application patch detection and kernel patch detection.

A. Application Patch Detection

Several empirical studies [35], [36] have analyzed the processes of fixing bug for the real-world vulnerabilities and summarized some characterizations of patches. BSCOUT [37] leverages the whole patch to detect the patches of Java executables. PATCHECKO [4] determines whether the target from Android libraries is a vulnerable function or a patched function by comparing the similarities of static function features and dynamic semantic features. SPAIN [14] identifies the security-related patches by analyzing the semantic differences between changed and corresponding unchanged binary blocks. The purpose of our work is to identify whether the target binary function has been patched. BINXRAY [12] generates the patch signatures from binary changed blocks that are obtained by a block mapping algorithm. Then BINXRAY detects the binary patched function by the similarity of the signatures. However, high compiler optimization levels introduce lots of binary changes unrelated to patches, which makes the accuracy of BINXRAY decrease significantly.

B. Kernel Patch Detection

FIBER [28] translates the most representative code changes into a binary signature and searches for the pattern in the target function of Android kernel images. PDIFF [33] generates the semantic summaries of patch-affected paths from the target binary functions of downstream kernels, which are often changed by code customization and non-standard building configurations. Then PDIFF determines whether the target function is patched based on the semantic summary similarity between the target function and the functions of its mainstream version. PDIFF achieves high accuracy and significantly reduces the false-negative rate compared to FIBER.

VI. CONCLUSION

This work proposes PMatch, a method based on code semantic similarity to detect the patched binary functions. PMatch leverages a novel unsupervised sentence embedding approach SIF in NLP, to generate the semantic feature vectors of code blocks. Then PMatch matches the patch-affected code snippets with the target blocks obtained by function diffing. The comparative test demonstrates that the unsupervised approach is more accurate, efficient, and scalable than the deep learning-based supervised method. We collect 101 CVEs and compile 304 binary programs with several different optimization levels for evaluation. The results show that PMatch outperforms the state-of-the-art work in the accuracy of detecting the patched functions at high optimization levels. Besides, at the O3 high optimization level, PMatch achieves an accuracy improvement of over 20%.

ACKNOWLEDGEMENT

This work is partly supported by National Key Research and Development Program of the Ministry of Science and Technology (Grant No.2020YFB1805405), National Natural Science Foundation of China Joint Fund Project (Grant No.U1766215), and Industrial Internet Innovation and Development Project (Grant No.KFZ0120200004).

REFERENCES

[1] "Heartbleed bug," https://heartbleed.com/, last accessed 10 May 2021.

[2] "Cve program report for q4 calendar year 2020," https://cve.mitre.org/blog/January262021_CVE_Program_Report_for_Q4_Calendar_Year_2020.html, last accessed 10 May 2021.

[3] S. Haefliger, G. Von Krogh, and S. Spaeth, "Code reuse in open source software," *Management science*, vol. 54, no. 1, pp. 180–193, 2008.

[4] P. Sun, L. Garcia, G. Salles-Loustau, and S. Zonouz, "Hybrid firmware analysis for known mobile and iot security vulnerabilities," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 373–384.

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

[6] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[7] 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*, 2016, pp. 480–491.

[8] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 57–67.

[9] 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*, 2017, pp. 363–376.

[10] 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 *Network and Distributed Systems Security (NDSS) Symposium 2019*, 2019.

[11] Y. Hu, H. Wang, Y. Zhang, B. Li, and D. Gu, "A semantics-based hybrid approach on binary code similarity comparison," *IEEE Transactions on Software Engineering*, 2019.

[12] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 376–387.

[13] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 180–190.

[14] 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.

[15] 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.

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

[17] S. Arora, Y. Liang, and T. Ma, "A simple but tough-to-beat baseline for sentence embeddings," 2016.

[18] "Pmatch: Semantic-based patch detection for binary programs," https://github.com/LN-Curiosity/PMatch, last accessed 21 May 2021.

[19] "Cve - cve-2014-0195," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0195, last accessed 10 May 2021.

[20] "Diaphora : A free and open source program diffing tool," http://diaphora.re/, last accessed 10 May 2021.

[21] "Cve - cve-2016-2182," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2182, last accessed 10 May 2021.

[22] "Cve - cve-2015-3197," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3197, last accessed 10 May 2021.

[23] "Ida pro - hex rays," https://www.hex-rays.com/products/ida/, last accessed 10 May 2021.

[24] "Cve - common vulnerabilities and exposures," http://cve.mitre.org/index.html, last accessed 10 May 2021.

[25] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "Patchscope: Memory object centric patch diffing," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 149–165.

[26] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, pp. 3111–3119, 2013.

[27] S. Jean, K. Cho, R. Memisevic, and Y. Bengio, "On using very large target vocabulary for neural machine translation," in *ACL (1)*, 2015.

[28] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 887–902.

[29] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[30] "scikit-learn: machine learning in python," https://scikit-learn.org/stable/, last accessed 10 May 2021.

[31] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. LeCun, C. Moore, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, no. 04, pp. 669–688, 1993.

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

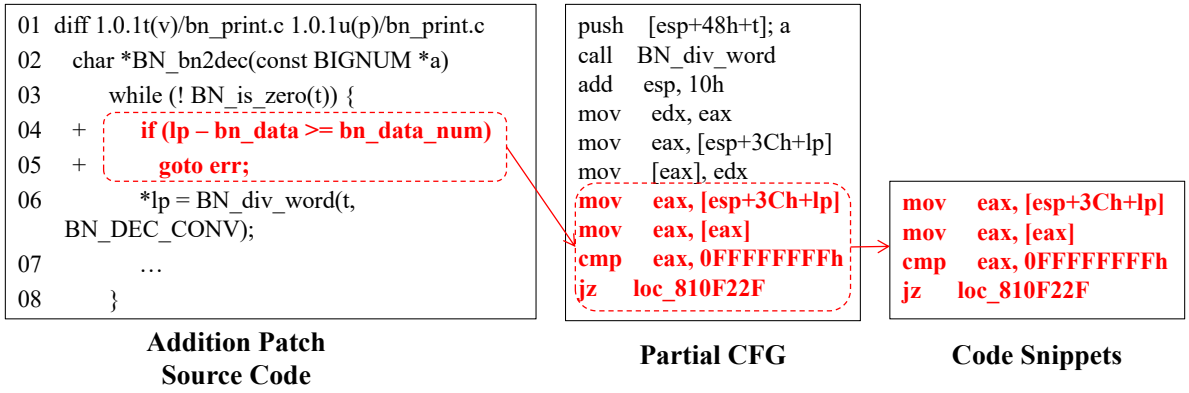
[33] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "Pdiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1149–1163.

[34] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

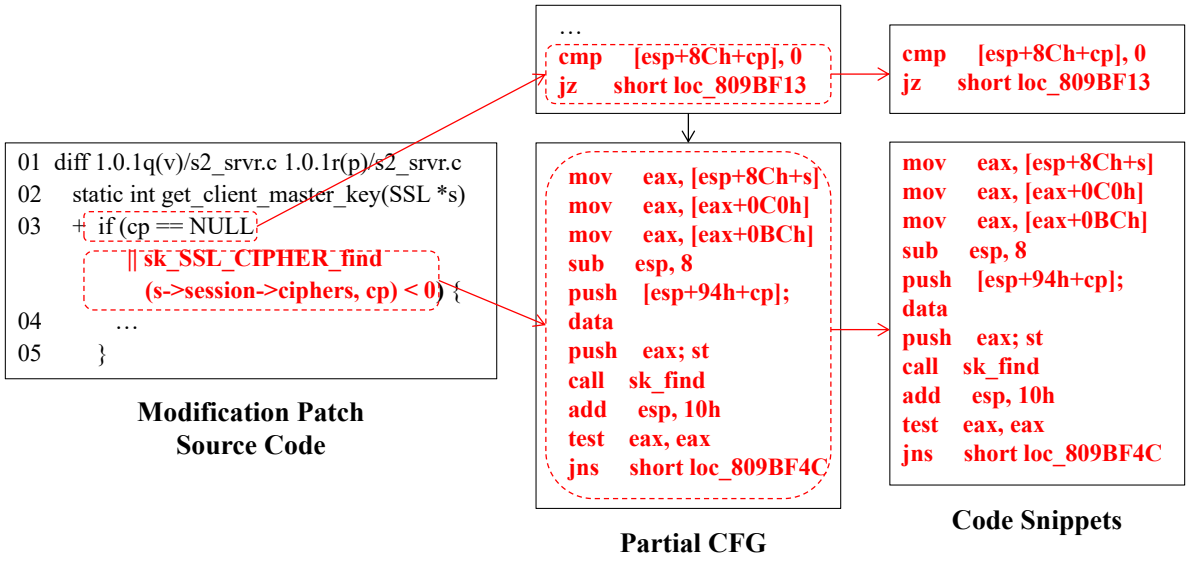
[35] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 913–923.

[36] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[37] J. Dai, Y. Zhang, Z. Jiang, Y. Zhou, J. Chen, X. Xing, X. Zhang, X. Tan, M. Yang, and Z. Yang, "Bscout: Direct whole patch presence test for java executables," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1147–1164.



(a) The code snippets for addition patch (of CVE-2016-2182).



(b) The code snippets for modification patch (of CVE-2015-3197).

Fig. 7: Two types of patch-affected code snippets.