Software Quality Journal (2019) 27:1045–1068
https://doi.org/10.1007/s11219-018-9435-5

CVSkSA: cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph

Dongdong Zhao[1] · Hong Lin[1] · Linjun Ran[1] · Mushuai Han[1] · Jing Tian[1] · Liping Lu[1] · Shengwu Xiong[1] · Jianwen Xiang[1]

Published online: 14 February 2019

Abstract
To prevent the same known vulnerabilities from affecting different firmware, searching known vulnerabilities in binary firmware across different architectures is crucial. Because the accuracy of existing cross-architecture vulnerability search methods is not high, we propose a staged approach based on support vector machine (SVM) and attributed control flow graph (ACFG) at the function level to improve the accuracy using prior knowledge. Furthermore, for efficiency, we utilize the k-nearest neighbor (kNN) algorithm to prune and SVM to refine in the function prefilter stage. Although the accuracy of the proposed method using kNN-SVM approach is slightly lower than the accuracy of the method using only SVM, its efficiency is significantly enhanced. We have implemented our approach CVSkSA to search several vulnerabilities in real-world firmware images. The experimental results show that the accuracy of the proposed method using kNN-SVM approach is close to the accuracy of the method using only SVM in most cases, while the former is approximately four times faster than the latter.

Keywords Firmware security · Cross-architecture · kNN-SVM · Bipartite matching

1 Introduction

In general, firmware refers not only to the interface combining the hardware with the software, but also to the software residing in the hardware. Firmware is an important part of IoT systems, the BIOS in computer systems, and the programs in extension ROM, as well as executable programs of common network devices, such as routers, switches, and webcams, all of which are typical firmware. However, similar to common software, firmware

✉ Jianwen Xiang
jwxiang@whut.edu.cn

[1] Hubei Key Laboratory of Transportation Internet of Things, School of Computer Science and Technology, Wuhan University of Technology, Wuhan 430070, Hubei, China

may also have vulnerabilities (referring to the weaknesses that can be exploited by attackers performing unauthorized actions, which introduce potential risk to IoT systems) Costin et al. 2014). ZoomEye's statistical report (Knownsec 2017) showed that 23% of more than 60,000 routers available for public search were affected by backdoor mechanisms in 2013. According to the report of OWASP (Open Web Application Security Project) in 2014, among the top 10 attacks on IoT systems, the attacks on software and firmware in embedded devices ranked ninth (Open web application secutity project 2016). With increasingly more security incidents occurring due to malicious firmware (Adelstein et al. 2002), there is a greater awareness of the importance of the vulnerability search in firmware, especially in cross-architecture scenarios.

Because obtaining the source code of most firmware is difficult, the known vulnerability search works mainly at the binary code level. Abundant approaches exist to search known vulnerabilities at the binary code level. However, most of these existing approaches either utilize dynamic analysis or are limited to the same architecture. The dynamic analysis on the firmware generally needs the specific devices or the simulation environment to run the target binary firmware. Additionally, there are several strict requirements for the target code to execute, so it is laborious to apply the dynamic analysis to cross-architecture vulnerability search cases. Other approaches, such as the k-gram and sequence alignment of instructions, obtain the opcodes or instructions for analysis, but these approaches are closely related to the specific architecture; therefore, directly applying them to the known vulnerability search across different architectures is difficult.

Since Pewny et al. (2015) presented the pioneering work on a similarity comparison to the known binary vulnerabilities across different architectures, there are only few researches on cross-architecture known vulnerability search. Recently, one advanced method to search known vulnerabilities across different architectures was proposed by Eschweiler et al. (2016). They employed a prescreening method to screen out most of the dissimilar functions and then used the MCS (maximum common subgraph) algorithm to determine the true matching function among a few suspicious functions. Although the method is effective in the cross-architecture cases, its prescreening stage seems unreliable and might result in poor accuracy in some scenarios (e.g., the case in Section 3).

In this paper, to enhance the overall efficiency, we adopted a staged strategy for the vulnerability search in firmware. To obtain a small portion of the candidate functions quickly, we first used kNN to prune (similar to the work by Eschweiler et al. in 2016), and then used SVM to refine, which can result in a higher accuracy in the prescreening stage. Then, inspired by the work of Feng et al. (2016), we used bipartite matching to pick out the true matching functions from the suspicious functions that remained from the prescreening stage. The experimental results show that CVSkSA achieves good performance in vulnerability search.

This paper is a significant extension of the conference paper published at DSA 2017 (Lin et al. 2017). On the basis of the original paper, we further propose a hybrid method using a kNN-SVM approach to improve the efficiency considerably at an acceptable or negligible cost of accuracy. The idea is to use kNN for fast screening out of obvious non-candidates, and then, SVM is only applied to a small number of highly suspected functions. The kNN-SVM-based method can reduce the query time to a few times lower than that of our previous work only applying SVM (Lin et al. 2017), e.g., from 0.18 s to 0.032 s in the condition "ARM to MIPS", at the expense of slightly lower correctness, e.g., from 99.7 to 99.6%. The kNN-SVM-based method also outperforms other state-of-the-art approaches, i.e., Multi−MH (Pewny et al. 2015), Multi−k−MH (Pewny et al. 2015), and

discovRE (Eschweiler et al. 2016), in terms of overall performance. In summary, our major contributions are as follows:

1. We show a staged approach, CVSkSA, to search vulnerabilities in binary firmware across different architectures, which can take advantage of the knowledge that we already know about the vulnerabilities.
2. We perform some experiments on a baseline dataset and real-world firmware images. Compared with Multi−MH (Pewny et al. 2015), Multi−k−MH (Pewny et al. 2015), and discovRE (Eschweiler et al. 2016), the experimental results show that CVSkSA has not only a better accuracy in vulnerability search but also a higher efficiency.

The rest of this paper is organized as follows: Section 2 is the overview of the proposed approach. Section 3 mainly discusses the implementation of our approach, and compares it with the state-of-the-art approaches. Section 4 shows the experimental evaluation of our approach on real-world vulnerable functions and firmware images under realistic conditions. Section 5 presents some related works about recognizing the known vulnerabilities and some works about the hybrid method using kNN-SVM in other application fields. Section 6 mainly demonstrates some restrictions of our approach. Section 7 provides a summary of our work and presents the future work.

2 Approach overview

In this paper, we divide the framework of CVSkSA into three stages, as shown in Fig. 1.

In the first stage, we utilize IDA pro (Guifanow 2016) for preprocessing the firmware. After preprocessing, the firmware includes several binary files that consist of many functions after disassembly, so if we want to recognize a known vulnerability in firmware, we can inspect the suspicious functions in the binary files contained by the firmware. Preparing for similarity computation in the next two stages, we extract the function level and basic block level features of the functions, which vary slightly across different architectures. The function level features mainly refer to the basic properties of the functions, such as the set of strings, the number of function calls, etc. The basic block level features mainly refer to the ACFGs (attributed control flow graphs) of the functions. Compared with the function level features, the basic block level features can be used for more accurate similarity computation between functions, but the computing process is more costly.

In the second stage, the goal is to screen out the dissimilar functions and obtain a small portion of the similar functions to be inspected in the next stage. We propose two approaches to reach this goal. One approach is to use SVM directly for function prescreening. We first compute the similarity vector of the compared functions based on their function level features. With prior knowledge, we utilize training samples to build the SVM model. Then, we take the similarity vector as the input of the trained SVM model. The output of the model is a label identifying whether the compared functions are similar or not. In this way, we can correctly filter out most of the dissimilar functions. The other approach is to use the hybrid method kNN-SVM for function prescreening. Although we can obtain good accuracy by using only SVM, the efficiency of SVM is not very good. Thus, by taking advantage of the simple distance function of kNN, we can utilize kNN to obtain a smaller collection of functions quickly for a further check by SVM. Based on this hybrid method kNN-SVM, we can efficiently screen out most of the dissimilar functions and obtain fewer suspicious functions for the next step.

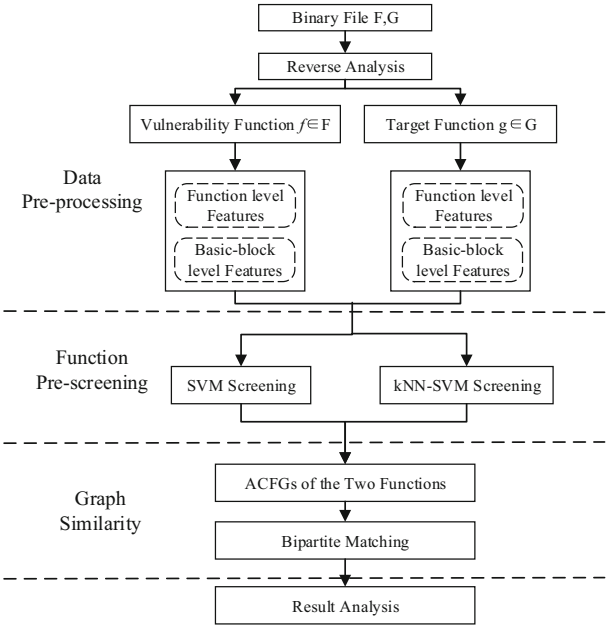


Fig. 1 The framework of CVSkSA

In the third stage, we consider the function-matching problem as a graph-matching problem, and we use the bipartite graph matching to inspect the suspicious functions based on their ACFGs.

3 Implementation of CVSkSA

In this section, we discuss the implementation of CVSkSA in detail.

3.1 Data preparation

3.1.1 Dataset I—baseline assessment

For the baseline assessment, we use OpenSSL v1.0.1f (2014) and BusyBox v1.21 (2013) to compile for three architectures, MIPS, x86, and ARM, as Pewny et al. (2015) and Eschweiler et al. (2016) have done, which generates about 18,000 functions with at least five basic blocks. The three chosen architectures are popular in IoT devices, and are applications whose source codes cannot be accessed. The binaries we compiled are all unstripped, so we can take the symbolic information, such as the function names, as the basis of the assessment.

3.1.2 Dataset II—real-world firmware dataset

To evaluate the practicability of our approach, we apply our approach to real-world vulnerable functions, i.e., c2i_ASN1_OBJECT, EVP_DecodeUpdate, X5 09_cmp_time, tls_decrypt_ticket, tls1_process_heartbeat, and hedwigcgi_main. The related firmware

datasets include OpenSSL 1.0.1.e (2013), DD−WRT r21676 (2013), Netgear ReadyNAS v6.1.6 (2014), DIR-815 (2014), DIR300 (2014), DIR600 (2013), and DIR-645 (2013).

3.2 Data preprocessing

IDA pro is a reverse parsing tool, which can transfer the binary code to assembly language source code and can support different processors and operating systems for the binary files. Because of the strong ability of IDA pro to handle the binary files across a vast number of architectures, we use this tool to disassemble the firmware and use its API to write plug-ins for feature extraction of the functions.

We mainly extract features at two levels: the function level and the basic block level. We extract the same type of features as Eschweiler et al. (2016), which have been proven to be robust for prefiltering functions. The detailed features are shown in Table 1. For example, the function calls and incoming calls refer to how often a function is called and how many functions it calls, the basic blocks and edges are used to describe the CFG, the offspring refers to the number of children nodes, and the betweenness refers to the centrality in a graph. The function-level features are used for the preliminary screening, and the basic block level features mainly serve for graph matching of the ACFGs. More details about the features can be found in the work by Eschweiler et al. (2016).

3.3 Prescreening of functions based on SVM

SVM is a supervised learning algorithm, which is often used for classification and regression analysis (Ukil 2002). With prior knowledge from a small amount of training samples, SVM can be applied to classify data well. Therefore, to screen out most of the dissimilar functions and promote the accuracy of the function prescreening stage, we use SVM on

Table 1 Two-level features used in CVSkSA

Type	Feature name
Function level	No. of function calls
	No. of logic instructions
	No. of redirection instructions
	No. of transfer instructions
	Size of local variable
	No. of basic blocks
	No. of edges
	No. of incoming calls
	No. of instructions
Basic block level	No. of string constants
	No. of function calls
	No. of control transfer instructions
	No. of arithmetic instructions
	No. of incoming calls
	No. of instructions
	No. of offspring
	Betweenness

function level features to recognize a subset of suspicious functions to be inspected in the next stage. We mainly divide the functions into two categories. One is similar to the vulnerable function, the other is dissimilar to the vulnerable function. Based on SVM, we can accurately filter out the dissimilar functions and retain the similar functions that are most likely vulnerable. The process of SVM is given as follows.

3.3.1 Computing similarity vectors

Given the compared functions f and g, each function has a nine-dimensional numeric feature vector, described above, and we calculated the similarity between the compared feature vectors on each dimension to obtain the similarity vector. We input the similarity vector into the SVM which has been trained by the training set, and the output of the SVM was a label "1" or "0", indicating whether the two functions are similar or not. We used the same approach used by Chang et al. (2016) to compute the similarity of the ith (i=1...9) dimension as follows:

$$sim = \begin{cases} c & f_i = 0 \text{ and } g_i = 0 \\ 1 - \frac{|f_i - g_i|}{max(f_i, g_i)} & \text{otherwise} \end{cases} \quad (1)$$

In (1), c is a constant between 0 and 1, which is taken as the same in each dimension, and f_i and g_i are the numerical values of the ith dimension of the feature vectors of f and g.

3.3.2 Construction of training samples

Suppose that we know the vulnerable function is on a particular architecture and the architectures of the suspicious functions are also known. We can take full advantage of the previous knowledge we have to obtain the higher accuracy in recognizing the known vulnerability on a specific architecture. Thus, we construct the training samples for the SVM model as follows:

Taking the construction of training samples across ARM to MIPS for example, we compile the source code of Busybox v1.21 on ARM and MIPS separately. After compiling, we obtain two unstripped binary files A and M. If f is a function in A, then there is a function f' with the same name as the function f in M. We take the similarity vector between the compared functions f and f' as the positive training samples with the label "1". Then, randomly selecting ten functions g_i in M, which are all different from the function f in A, we take the similarity vector between the compared functions f and g_i as negative training samples with the label "0". The proportion of positive and negative training samples is 1:10.

3.3.3 Evaluation of the prescreening of functions based on SVM

We take OpenSSL v1.0.1f as the test sample, and we compile the source code of OpenSSL across ARM, MIPS, and x86 separately to obtain three unstripped binaries. The results are shown in Table 2. The kNN-based method simply obtained several closest functions to the vulnerable function based on the Euclidean distance. Although it is easy to implement kNN, the method lacks credibility (Feng et al. 2016). Different from kNN, SVM can be trained and can learn from the prior knowledge to improve the classification accuracy. Once the SVM model is trained, we can process the new data more easily and accurately.

As shown in Table 2, "From→To" refers to the notion that we recognize a known vulnerability from one architecture to another architecture, similar to the work by Pewny et al. (2015). This table presents the ratios of the true matching functions, the average number of

Table 2 Evaluation of SVM and kNN with OpenSSL

From→To	Avg. no. of candidates		Percent correct (%)		Average query time (s)	
	kNN	SVM	kNN	SVM	kNN	SVM
ARM→MIPS	128	110	99.1	99.7	0.0004	0.018
MIPS→ARM	128	110	98.9	99.7	0.0003	0.018
ARM→x86	128	109	57.1	99.1	0.0003	0.020
x86→ARM	128	109	88.9	99.1	0.0002	0.021
MIPS→x86	128	77	58.6	99.4	0.0002	0.018
x86→MIPS	128	77	95.3	99.4	0.0003	0.019

suspicious functions, and the average query time of kNN and SVM in each deriving condition. We can see that, although the average query time of kNN is much shorter than that of SVM, the accuracy of SVM is much higher than that of kNN. The accuracy of SVM is greater than 99% in every architecture combination, while the accuracy of kNN is not stable; the lowest is only 57.1%. We can also observe that the accuracies in conditions "ARM to x86," and "MIPS to x86" are the worst. That is, when we use the test data in x86 to create the k-d tree, the search results are poor; the reason may be that kNN is sensitive to the local structure of data, and it may not be suitable for the situation that involves x86. In addition, the average number of suspicious functions obtained by SVM is smaller than those obtained by kNN ($k = 128$).

As shown in Fig. 2, the ratio of the true matching functions from x86 to MIPS changes with the k values. If the k is smaller, the accuracy is lower; otherwise, the accuracy is higher. However, even when the k is large enough, the accuracy of kNN is close to that of SVM.

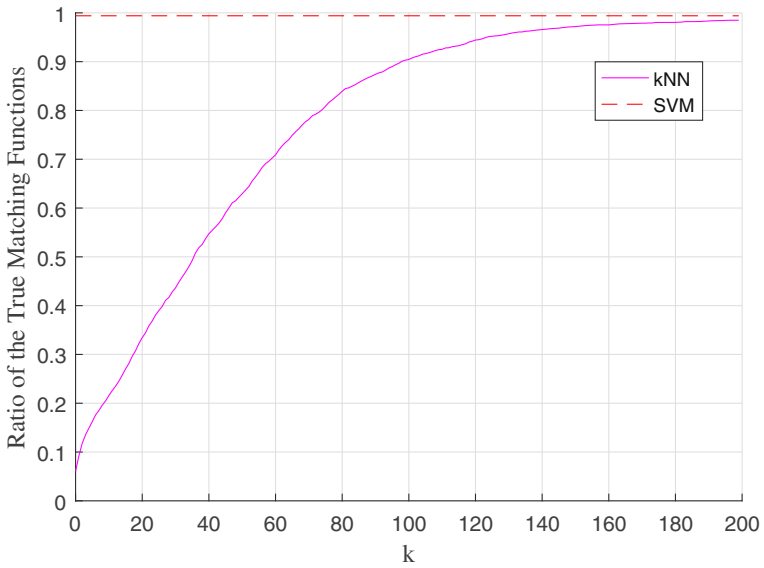


Fig. 2 The ratio of the true matching functions for different k values in the condition of "x86 to MIPS"

Obviously, SVM has a higher accuracy than kNN and does not require knowing how to choose the k value.

3.4 Prescreening of functions based on kNN-SVM

In general, the search code base of firmware is large; thus, if we want to apply the method to real-world scenarios, the efficiency is a critical factor. If we can filter out most of the dissimilar functions in the firmware and only leave a small fraction of suspicious functions to be inspected focally, the efficiency will be greatly improved. In Section 3.3.3, the experimental results show that kNN has a higher efficiency, and SVM has a higher accuracy. Taking the efficiency and accuracy into account, we first use kNN to obtain k nearest functions. Then, based on these functions, we employ SVM to recognize a smaller subset of suspicious functions to be inspected in the next stage.

3.4.1 Pruning with kNN and refining with SVM

It is well-known that kNN is one of the simplest methods among machine learning algorithms, which is referred to as lazy learning. Because of its simplicity and high efficiency, we utilize it to obtain the k similar functions from the large function base, which can save a considerable amount of time being spent on the unnecessary functions.

Similar to the work by Eschweiler et al. (2016), we use k-d trees to implement the kNN algorithm. The space and time complexities of k-d tree are $O(n)$ and $O(logn)$, respectively, which are better than those of the linear index and hierarchical k-means, and k-d tree is efficient and beneficial for repeated queries on the same data structure (Eschweiler et al. 2016). We first normalize the numeric value on each dimension of the function-level feature vectors from the firmware. Then, we utilize these normalized feature vectors to create several k-d trees for searching in parallel. Based on kNN, we can finally obtain k similar suspicious functions to be refined by SVM.

After kNN processing, there are k functions left to be inspected by SVM. The goal is to further narrow the range of suspicious functions. We use the approach described in subsection 3.3 to construct the training samples for the SVM model and utilize the trained model to recognize the suspicious functions. By training, we can obtain a separation hyperplane in SVM, which is a good boundary to partition the dissimilar and similar functions. Thus, we can correctly refine the range of suspicious functions for the next stage.

3.4.2 Determining the appropriate value of k

In the kNN-SVM hybrid method, how to choose the appropriate value of k is an important issue. Obviously, if the value of k is too small, the real vulnerable function may not be in the k functions and the accuracy will decline sharply. In contrast, if the value of k is too large, there will be too many functions to address. Therefore, we should choose a suitable value of k to guarantee the accuracy as well as the efficiency.

To obtain the appropriate value of k, we still take OpenSSL v1.0.1f as the test sample. As shown in Fig. 3, the accuracy increases with k in the three conditions. In most cases, when the value of k is greater than 200, the accuracy approaches 1 and tends to be stable. Thus, in the conditions of "ARM to MIPS," "MIPS to ARM," "x86 to ARM," and "x86 to MIPS", the suitable value of k is approximately 200. However, in the conditions of "ARM to x86" and "MIPS to x86", the accuracies are still not high when the value of k is 200. The appropriate k value for these two conditions is greater than 500. The reason for the low accuracies in the

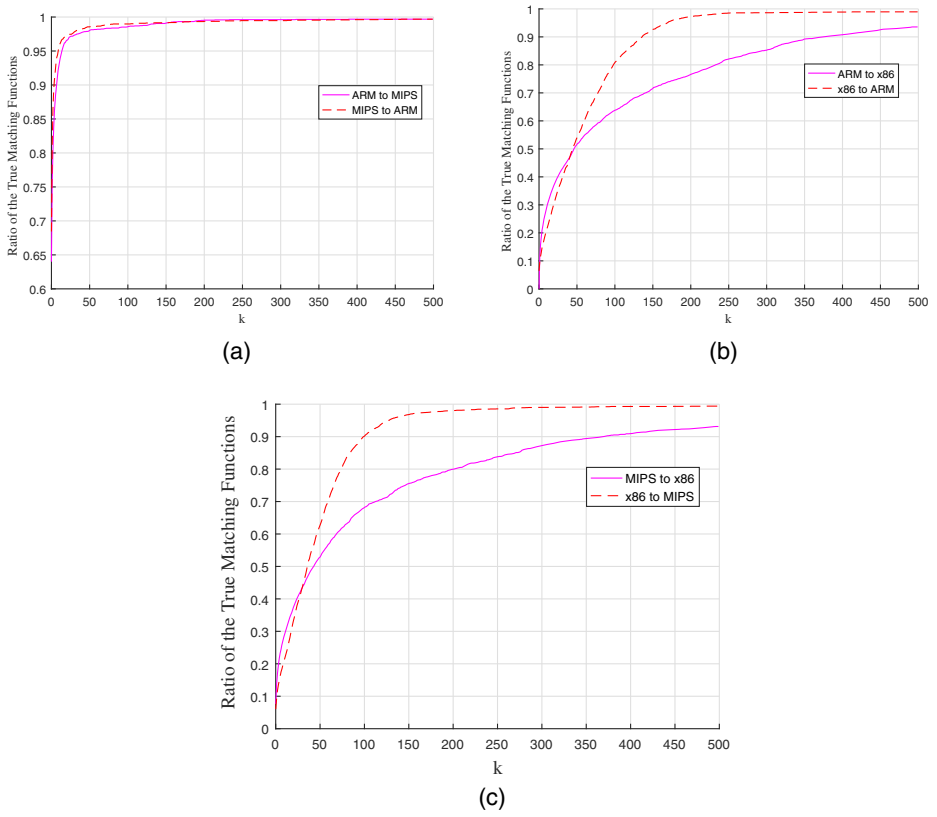


Fig. 3 The ratio of the true matching functions for different k values in the conditions of "ARM to MIPS," "MIPS to ARM," "ARM to x86," "x86 to ARM," "MIPS to x86," and "x86 to MIPS"

conditions of "ARM to x86" and "MIPS to x86" with k at 200 may be that the features "the size of local variable" and "the number of transfer instructions" degrade the accuracy when x86 is included. We find that when we remove the features "the size of local variable" and "the number of transfer instructions" from the feature vector, the accuracies are improved considerably in the conditions of "ARM to x86," "MIPS to x86," "x86 to ARM," and "x86 to MIPS". In particular, for the conditions "ARM to x86" and "MIPS to x86", the improved accuracies with k at 200 are close to the previous accuracies with k at 500, while removing these two features has little impact on the accuracies for the conditions "ARM to MIPS" and "MIPS to ARM". This is an interesting issue and we will investigate it to find out the underlying reasons in our future work.

3.4.3 Evaluation of the prescreening of functions based on kNN-SVM

As in the previous evaluation, we continue to use the OpenSSL v1.0.1f as the test sample. We present the results of SVM and kNN-SVM in Table 3. Similarly, the table shows the ratios of the true matching functions in different conditions and presents the average number of suspicious functions in each deriving condition. Taking all deriving conditions into consideration, we set the value of k to 500.

Table 3 Evaluation of SVM and kNN-SVM with OpenSSL

From→To	Avg. no. of candidates		Percent correct(%)	
	SVM	kNN-SVM	SVM	kNN-SVM
ARM→MIPS	110	19.3	99.7	99.6
MIPS→ARM	110	19.4	99.7	99.6
ARM→x86	109	60.5	99.1	93.7
x86→ARM	109	19.2	99.1	99.0
MIPS→x86	77	24.0	99.4	93.1
x86→MIPS	77	13.4	99.4	99.4

As shown in Table 3, in most deriving conditions, the accuracy of kNN-SVM is close to that of SVM. In the worst case, "ARM to x86", the accuracy of kNN-SVM is 93.7%, which is still better than the accuracy 53.7% achieved by Eschweiler et al. (2016) using only kNN. In addition, we can see that the accuracies for conditions, "ARM to x86" and "MIPS to x86" are the worst, similar to results in Table 2. This observation indicates that kNN might be not suitable for vulnerability search in the x86 platform.

Although combining kNN and SVM decreases the accuracy, the efficiency is improved. The average search time of SVM is 0.022 s, while that of kNN-SVM is only 0.0035 s (shown in Table 4). In addition, as shown in Fig. 4, we can see that the average number of candidate functions for kNN-SVM is much smaller than that for SVM. It is noted that the candidate functions are mainly inspected by bipartite matching in the third stage; thus, the smaller the number is, the higher the efficiency will be.

In summary, the kNN-SVM only makes some compromise in accuracy, while the efficiency is enhanced not only in the prescreening stage but also in the stage of graph similarity computation.

In addition, Table 4 mainly presents a global comparison of the kNN, SVM, and kNN-SVM approaches in terms of accuracy and efficiency. The k value of kNN is 128 and that of kNN-SVM is 500. As shown in Table 4, we can observe that the accuracy of SVM is the best among the three approaches. However, the efficiency of SVM is the worst. In contrast, the accuracy of kNN is the worst, but the efficiency of kNN is the best. Compared with SVM, kNN-SVM has a lower accuracy but a higher efficiency. Thus, according to different application scenarios, we can choose the most suitable approach from these three algorithms.

Table 4 Evaluation of kNN, SVM and kNN-SVM with OpenSSL

From→To	Percent correct(%)			Average query time (s)		
	kNN	SVM	kNN-SVM	kNN	SVM	kNN-SVM
ARM→MIPS	99.1	99.7	99.6	0.0004	0.018	0.0032
MIPS→ARM	98.9	99.7	99.6	0.0003	0.018	0.0032
ARM→x86	57.1	99.1	93.7	0.0003	0.020	0.0035
x86→ARM	88.9	99.1	99.0	0.0002	0.021	0.0037
MIPS→x86	58.6	99.4	93.1	0.0002	0.018	0.0031
x86→MIPS	95.3	99.4	99.4	0.0003	0.019	0.0033

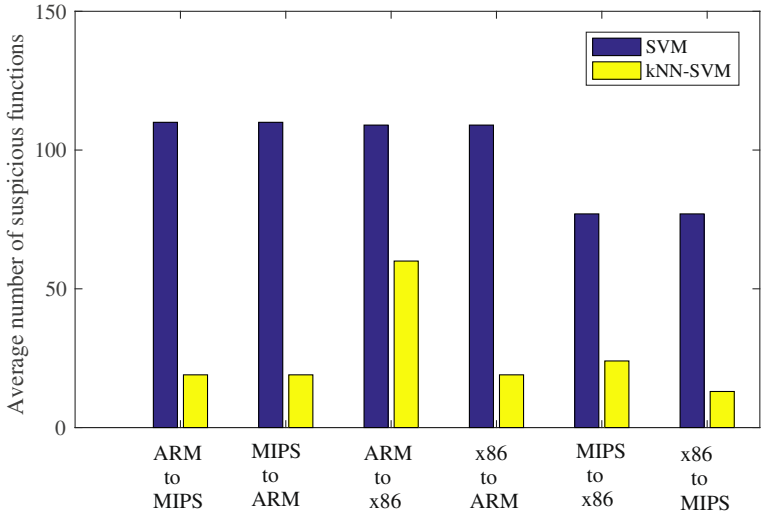


Fig. 4 Search results based on SVM and kNN-SVM in different deriving conditions

3.5 Graph similarity calculation based on bipartite matching

In this stage, our task is to pick out the true matching functions from the suspicious functions based on their ACFGs. In essence, the ACFG is exactly the CFG with a series of features that are helpful in the process of graph matching (Feng et al. 2016). Therefore, we transfer the function-matching problem into a graph-matching problem. There are several methods to handle this problem, such as the MCS used by Eschweiler et al. (2016) and bipartite matching used by Feng et al. (2016). It is demonstrated that the bipartite matching appears to be a better choice than the MCS. Therefore, we chose the bipartite matching to calculate the graph similarity between the suspicious functions' ACFGs and the vulnerable functions' ACFGs, which assists us in determining the true matching functions.

3.5.1 Bipartite matching for ACFGs

Bipartite graph is a graph whose vertices can be divided into two disjoint and independent sets (Diestel 1997). Because bipartite matching is concerned with a bipartite graph, as shown in Fig. 5, we take ACFG G_1 and ACFG G_2 as one part of the bipartite graph, similar to the work by Feng et al. (2016). Given the combined bipartite graph $G = (V, E)$, we have the following: V is the set of all nodes in G_1 and G_2, and E is the set of all edges in G_1 and G_2. As already mentioned, the nodes refer to the basic blocks in ACFG, the edges refer to the links of every associated basic blocks, and each edge is assigned a specific cost to measure the matching distance between the basic blocks. There are many matching solutions from the nodes in G_1 to the nodes in G_2. However, we only need to obtain the matching solution with the minimum distances. We use the Kuhn-Munkres algorithm (Bourgeois and Lassalle 1971) to inspect around the combined bipartite graph and then obtain the minimum matching distances.

In our approach, the matching distance between the associated basic blocks is calculated according to their feature vectors. We use the same approach as Feng et al. (2016) to obtain

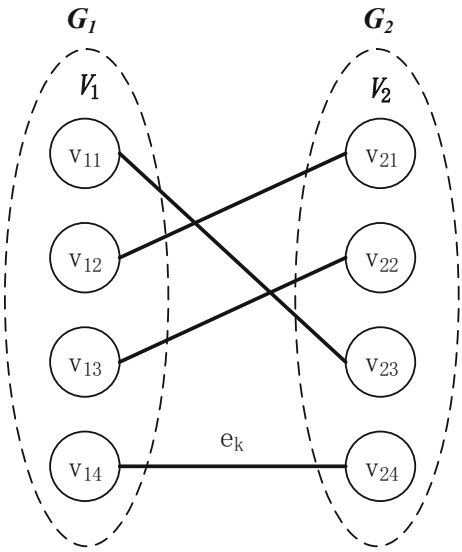


Fig. 5 Bipartite graph matching

the matching distance as follows:

$$d(v_1, v_2) = \frac{\sum_i \omega_i |\alpha_{1i} - \alpha_{2i}|}{\sum_i \omega_i max(\alpha_{1i}, \alpha_{2i})} \tag{2}$$

In (2), α_{1i} is the numeric value on the ith dimension of the feature vector of block v_1. Similarly, α_{2i} is the numeric value on the corresponding dimension of the feature vector of block v_2. ω_i represents the weight we have assigned to each dimension of the feature vector. We assign the detailed weight values similar to Eschweiler et al. (2016), which can maximize the distance between different ACFGs and minimize the distance of the equivalent ACFGs (Feng et al. 2016). The detailed weight values are shown in Table 5.

Table 5 The weight of the basic block level features

Feature name	Weight(ω)
No. of string constants	11
No. of function calls	66
No. of control transfer instructions	150
No. of arithmetic instructions	56
No. of incoming calls	66
No. of instructions	42
No. of offspring	199
Betweenness centrality	31

After calling the Kuhn-Munkres algorithm, we can obtain the minimum matching distances between the compared graphs. Next, we calculate the similarity of the compared graphs G_1 and G_2 as follows:

$$\psi(G_1, G_2) = 1 - \frac{d(G_1, G_2)}{min(d(G_1, \emptyset), d(G_2, \emptyset))} - p \times N \tag{3}$$

In (3), $d(G_1, G_2)$ is the minimum distance between the compared graphs. $\emptyset$ is the ACFG with all the feature values being zero, which has the same number of nodes as the matching graph. p is the penalty coefficient, and N is the difference between the numbers of basic blocks in the two compared ACFGs. If N is large to a certain degree, we can directly determine that the two compared ACFGs are not matching.

3.5.2 Evaluation of graph similarity based on bipartite matching

We continue our evaluations with OpenSSL v1.0.1f as the test sample to assess the graph similarity based on bipartite matching. We obtain three unstripped binaries, A, M, and X, after compiling the source code of OpenSSL across the three architectures we chose. Considering "ARM→x86" as an example, given a function in A, we calculate the graph similarity between the given function and the suspicious functions in X; then, we can obtain the true matching function in X and its rank. Top x refers to the percentage of the functions at rank x among all the suspicious functions. We used top 1, top 10 and top 100 (Pewny et al. 2015) as the assessment criteria. The result is shown in Table 6.

As shown in Table 6, the result for the condition "ARM to x86" is the best, for which the three top indicators are 91.18%, 97.99%, and 99.15%. We can observe that the results are not good for all cases that contain the MIPS architecture. However, even though the results are not very good in this situation, the top 1 indicator is still above 61.5%, and on our test dataset, the top 1 indicator is always better than that of the MCS used by Eschweiler et al. (2016). It indicates that the bipartite matching is more suitable than MCS for calculating graph similarity.

We also take the same test samples to evaluate CVSkSA, which uses both kNN-SVM and bipartite matching. Since there is exactly one true matching function for each query in our experiments, the number of false positives and the number false negatives are the same; and thus, the precision, recall and F1-measure are the same as well. Therefore, we only show the precision of CVSkSA in Fig. 6. As shown in Fig. 6, k means that we take the candidate functions on the top k as positives. We can see that the precision of CVSkSA increases with the value of k in all deriving conditions, and in the conditions of "ARM to x86" and "MIPS to x86", the results are not as good as in the other deriving conditions. The reason is that

Table 6 Top X of OpenSSL on different architectures

From→To	Top 1 (%)	Top 10 (%)	Top 100 (%)
ARM→MIPS	73.93	93.60	98.69
MIPS→ARM	80.17	95.46	98.66
X86→ARM	82.05	97.57	99.33
ARM→x86	91.18	97.99	99.15
MIPS→x86	61.56	89.64	99.01
x86→MIPS	64.34	91.65	98.48

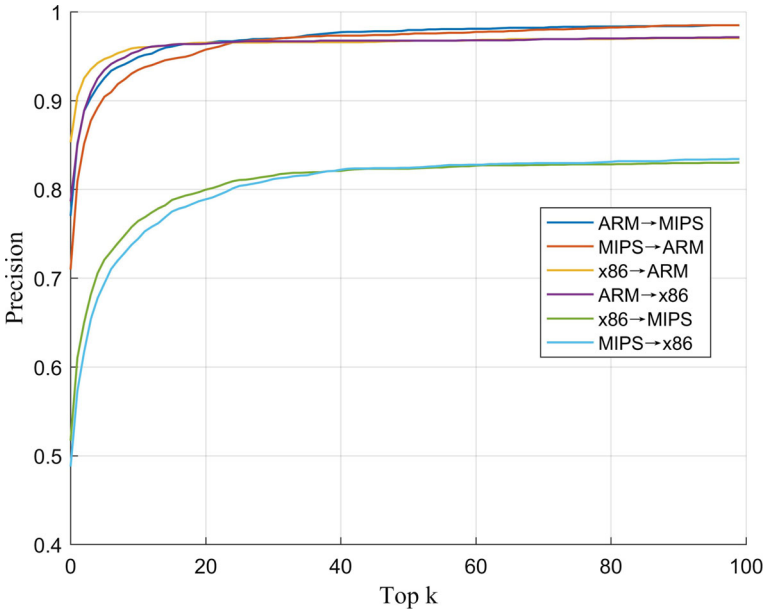


Fig. 6 Precision for different k values

kNN does not perform as well in the conditions of "ARM to x86" and "MIPS to x86" as in the other deriving conditions.

4 Experimental evaluation

In this section, we mainly apply the proposed method CVSkSA to real-world vulnerable functions and firmware images and compare it with other approaches. The experimental results show the effectiveness of CVSkSA.

We use IDA pro v6.8 to reverse and parse the binaries, and with its API, we employ Python script to extract the two levels of features and ACFGs of the functions in the binaries. SVM was implemented by the toolbox of Matlab R2016a.

4.1 Vulnerabilities in OpenSSL

We gather four real-world vulnerable functions in OpenSSL, as shown in Table 7. For convenience, we assign them IDs 1 to 4. We use CVSkSA to recognize the four known vulnerable functions across different architectures.

As shown in Table 8, "Rank" refers to the rank of the vulnerable function among the candidates. We can see that the ranks of the four vulnerable functions in OpenSSL obtained by our approach CVSkSA are all 1 across the three architectures, and most of the numbers of suspicious functions (i.e., the candidates in Table 8) with regard to the given vulnerable function are less than 70; the minimum is only 1.

In addition, we also recognize the vulnerable function with ID 4 from the three architectures that we chose to the MIPS-based firmware DD-WRT (r21676) (2013) separately.

Table 7 Four vulnerable functions in OpenSSL

Vulnerability function	CVE number	ID
c2i_ASN1_OBJECT	CVE-2014-3508	1
EVP_DecodeUpdate	CVE-2015-0292	2
X509_cmp_time	CVE-2014-3567	3
tls_decrypt_ticket	CVE-2014-1959	4

When we employ the vulnerable function from the given architecture to recognize the suspicious functions to DD-WRT, we can always successfully find the true matching function in the firmware.

4.2 Overflow vulnerabilities in firmware images

We also evaluate our approach in recognizing known overflow vulnerabilities which are common in IoT devices. We take the D-Link router firmware as the test dataset, which is a widely used router. We chose four D-Link router firmware, DIR-815 (2014), DIR-300 (2014), DIR-600 (2013), and DIR-645 (2013), which have the same overflow vulnerability (SAP10008), as D-Link officially announced. The function that causes the overflow vulnerability is hedwigcgi_main().

As shown in Table 9, we recognize the vulnerable function hedwigcgi_main() from one router firmware to another. The ranks of hedwigcgi_main() obtained by CVSkSA in four firmware are almost 1, which demonstrates that we can recognize the vulnerability correctly.

The experimental results demonstrate that our approach can identify the vulnerable function in the given firmware correctly. It is worth mentioning that the four D-Link firmware images are all based on MIPS. Although our approach aims at recognizing known vulnerabilities across different architectures, it could also be used for vulnerability search on the same architecture.

4.3 Comparing CVSkSA with other approaches

In this section, we compare CVSkSA with Multi-MH (Pewny et al. 2015), Multi-k-MH (Pewny et al. 2015), and discovRE (Eschweiler et al. 2016). We use the same dataset as the test samples, including OpenSSL 1.0.1.e (2013), DD−WRT r21676 (2013), and Netgear

Table 8 Search results of four vulnerable functions

From→To	CVSkSA							
	No. of candidates				Rank			
	1	2	3	4	1	2	3	4
ARM→MIPS	10	24	48	59	1	1	1	1
MIPS→ARM	7	21	39	61	1	1	1	1
ARM→x86	2	26	1	32	1	1	1	1
x86→ARM	2	12	1	9	1	1	1	1
MIPS→x86	37	17	14	43	1	1	1	1
x86→MIPS	13	11	13	24	1	1	1	1

Table 9 The rank of "hedwigcgi_main()" obtained by CVSkSA

From \ To	DIR-815	DIR-300	DIR-600	DIR-645
DIR-815	–	1	1	1
DIR-300	1	–	1	1
DIR-600	1	1	–	1
DIR-645	1	1	1	–

ReadyNAS v6.1.6 (2014), all of which contain the Heartbleed bug (CVE−2014−0160) functions tls1_process_heartbeat (TLS) and dtls1_process_heartbeat (DTLS) (Choue 2017).

The ranks of TLS and DTLS for the four functions are shown in Table 10. For example, "1;2" means that CVSkSA ranks TLS and DTLS at 1 and 2, respectively. We can see that all the ranks obtained by CVSkSA are 1 or 2 in each deriving condition, while Multi-MH, Multi-k-MH, and discovRE cannot achieve it. Regarding efficiency, the normalized average query time of CVSkSA is only 0.097 ms, which is approximately four times faster than discovRE and three or four orders of magnitude faster than Multi-MH, and Multi-k-MH. In summary, CVSkSA can be well applied to vulnerability search across different architectures.

5 Related work

Different from the approaches used for finding undiscovered vulnerabilities, the target of our work is to derive the known vulnerability on other different architectures by function matching; thus, we mainly discuss the related work on the approaches for recognizing

Table 10 Results for vulnerability search using different approaches

From→To	Multi-MH		Multi-k-MH		discovRE		CVSkSA	
	TLS	DTLS	TLS	DTLS	TLS	DTLS	TLS	DTLS
ARM→MIPS	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
ARM→x86	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
ARM→DD-WRT	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
ARM→ReadyNAS	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
ARM→x86	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
MIPS→ARM	2;3	3;4	1;2	1;2	1;2	1;2	1;2	1;2
MIPS→x86	1;4	1;3	1;2	1;3	1;2	1;2	1;2	1;2
MIPS→DD-WRT	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
MIPS→ReadyNAS	2;4	6;16	1;2	1;4	1;2	1;2	1;2	1;2
x86→ARM	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
x86→MIPS	1;7	11;21	1;2	1;6	1;4	1;3	1;2	1;2
x86→DD-WRT	70;78	1;2	5;33	1;2	1;2	1;2	1;2	1;2
x86→ReadyNAS	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2
Query normalized avg. time (s)	0.3 s		1.0 s		4.1×10^{-4} s		9.7×10^{-5} s	

known vulnerabilities, while the works, such as Rozzle (Kolbitsch et al. 2012) and Driller (Stephens et al. 2016), that contribute to the discovery of unknown vulnerabilities, are not within our discussion.

5.1 Vulnerability search at the source code level

Using code similarity to search known vulnerabilities is a common approach. There exist some works that investigate code similarity at source code level. Kamiya et al. (2002) proposed a token-based tool to recognize similar source code fragments. In the same way, CP-Miner (Li et al. 2004) employed the token sequence to search similar source code across scalable software. Other approaches, such as the approaches proposed by Jiang et al. (2007) and Yamaguchi et al. (2014), took advantage of the abstract syntax tree to find out the copy-pasted code. Bellon et al. (2007) even evaluated six clone approaches in various aspects. There are also some systems specifically designed for detecting unpatched similar codes, such as the one proposed by Jang et al. (2012).

5.2 Vulnerability search at binary level

Due to the difficulties of obtaining the source code of most firmware, it is important to conduct the known vulnerability search at the binary level. However, the lack of symbolic information in binaries makes it rather complicated to search the vulnerabilities.

In the early phase, the works mainly focused on the approaches based on sequence comparison of the instructions (David and Yahav 2014) or the bitstream (Kolter and Maloof 2004; Myles and Collberg 2005). For the latter, there are also some approaches to assess the semantics of the binary code. Binhunt (Gao et al. 2008) and its enhancing project iBinHunt (Ming et al. 2012) used symbolic execution and the theorem prover to capture the semantics of the basic blocks. Expose (Ng and Prakash 2013) is a tool that can utilize symbolic execution and syntactic matching techniques for binary code reuse. Based on the semantic equivalence, the two approaches are easily affected by small changes in code, so it is difficult to use them in a vulnerability search. Moreover, they are only designed for a single architecture, which is unable to derive the known vulnerability across different architectures. Afterward, BINJUICE (Lakhotia et al. 2013), BINHASH (Jin et al. 2012) and TEDEM (Pewny et al. 2014) were proposed. BINJUICE and BINHASH mainly operate on the basic block level, but they could not support multiple architectures. TEDEM employed the expression tree of the basic blocks to find vulnerabilities in binaries.

There also exist a few methods for known vulnerability search across various architectures at the binary level. Pewny et al. (2015) presented pioneering work to recognize the known vulnerabilities across various architectures. They utilized VEX to weaken the influence of the different architectures, and then, they employed I/O behaviors to grasp the semantics on code similarity computing. They even used MinHash to speed up the search process; however, there are still some efficiency problems of the strategy (Eschweiler et al. 2016). Taking into account the efficiency problem, DiscovRE (Eschweiler et al. 2016) used a staged strategy to screen out most of the dissimilar functions and then obtained a small part of the suspicious functions to be inspected by graph matching. However, the prescreening strategy was demonstrated to be unreliable. Instead of performing the vulnerability search on CFGs originally, Feng et al. (2016) transformed the ACFGs into numerical vectors for similarity computing, which was inspired by the search methods in other areas. Chang et al. (2016) proposed a staged method VDNS based on a neural network and local calling structure matching to promote the efficiency of cross-architecture vulnerability search.

5.3 Vulnerability search in dynamic analysis

Different from the static analysis methods, dynamic analysis methods run the target code on specific devices or in simulated surroundings for the vulnerability search. Avatar (Zaddach et al. 2014) combined real-world embedded devices with a simulated environment to execute the suspicious code dynamically. Similar to our work, Egele et al. (2014) used BLEX to recognize the known vulnerable functions at the binary level. However, they did not analyze the function code statically, as we did, but executed each function under the simulated surroundings to obtain their runtime features. Due to its need for a specific environment, BLEX is not suitable for deriving the known vulnerabilities across different architectures. Moreover, it is only useful for the dynamic methods on firmware images in the early phase (Chen et al. 2016; Zaddach et al. 2014).

5.4 The application of kNN-SVM

kNN and SVM are widely used in the classification field. Because of the simplicity of kNN and the high performance of SVM in nonlinear classification, there are also many works on combining kNN and SVM to promote the efficiency. Hsu et al. (2005) utilized a two-stage SVM model for classification. By combining kNN in the filtering stage, they enhanced the performance of classification accuracy. Zhang et al. (2006) presented a combined method SVM-kNN for a visual category, which outperformed kNN and SVM separately. Tabrizi and Cavus (2016) applied a hybrid kNN-SVM model on Iranian license plate recognition. They used kNN for the first step to recognize the similar characters and SVM for the further classification, which attained a greatly improvement in recognition phase. Similarly, Yusof et al. (2016) proposed a hybrid kNN-SVM method to classify the DDoS attack.

6 Threats to validity

Although our approach is demonstrated to be able to recognize the vulnerable function across different architectures, there are still some threats to the validity of our approach. Inspired by the work of Feldt and Magazinius (2010), we show some threats to the validity of our approach as follows:

First, in our approach, the premise is that each of the vulnerabilities is caused by a specific function, that is, we apply CVSkSA at the function level. However, there are some cases that the vulnerabilities are raised by several functions that work together. There are also some cases in which the vulnerabilities are only raised by a fraction of the code or sometimes even by a constant string in the function. Although our approach cannot handle the cases described above, a mass of vulnerabilities can be located at one function which can be recognized by CVSkSA. The function level is sufficient for us to address most application scenarios.

Second, in our experiments, all of the test data we used contain the specific suspicious function that matches the known vulnerability; however, there are some cases in which some of the functions we find with the known vulnerabilities are not the true matching functions. These cases will be studied in our future work.

Third, in our approach, we do not take compiler optimization options into consideration. However, if we can recognize the compiler optimization options before vulnerability search, the accuracy would be improved. How to recognize the compiler optimization will be studied in our future work.

7 Conclusion and future work

In this paper, we demonstrate and implement a staged method, CVSkSA, to recognize the known vulnerabilities in firmware across different architectures (mainly ARM, MIPS, and x86). Compared to the previous work, our approach utilizes the knowledge already known about the vulnerability to achieve a higher accuracy in the prescreening stage. Furthermore, taking advantage of the simple distance function of kNN, we utilize kNN to quickly obtain a smaller set of suspicious functions for SVM to further filter out to enhance the efficiency. We also apply the approach to real-world firmware images. The results show that it could correctly recognize vulnerable functions, such as the overflow vulnerability, across the three architectures (i.e., ARM, MIPS, and x86).

In the future, we will apply other machine learning algorithms, such as neural network and decision tree, to the proposed method. By analyzing their advantages and disadvantages, we can choose the most appropriate approach according to the different application situations.

In addition, we plan to further explore the vulnerability search over dynamic link libraries. In this paper, we took the functions' similarity of the dynamic link libraries as a constant. By analyzing the library files where the suspicious functions reside, we could extract the features of those functions and reach a higher accuracy. We are also going to apply our approach to a large firmware image dataset in the future.

Funding information This work was partially supported by the National Natural Science Foundation of China (Grant No. 61672398, 61702386, 61806151), the Key Natural Science Foundation of Hubei Province of China (Grant No. 2017CFA012), the Major Technical Innovation Program of Hubei Province (Grant No. 2017AAA122), and the Applied Fundamental Research of Wuhan (Grant No. 2016010101010004).

References

Adelstein, F., Stillerman, M., Kozen, D. (2002). Malicious code detection for open firmware. In *Computer security applications conference* (pp. 403–412). IEEE.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E. (2007). Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, *33*(9), 577–591.

Bourgeois, F., & Lassalle, J.C. (1971). An extension of the munkres algorithm for the assignment problem to rectangular matrices. *Communications of the ACM*, *14*(12), 802–804.

Busybox v1.21 (2013). Accessed 5 Jan 2017.

Chang, Q., Liu, Z., Wang, M., Chen, Y., Shi, Z., Sun, L. (2016). Vdns:an algorithm for cross-platform vulnerability searching in binary firmware. *Journal of Computer Research & Development*, *53*(10), 2288–2298.

Chen, D.D., Woo, M., Brumley, D., Egele, M. (2016). Towards automated dynamic analysis for linux-based embedded firmware. In *Network and distributed system security symposium* (pp. 1–16).

Choue, A. (2017). On detecting heartbleed with static analysis. Accessed 5 Sept 2017.

Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., Antipolis, S. (2014). A large-scale analysis of the security of embedded firmwares. In *USENIX Security symposium* (pp. 95–110).

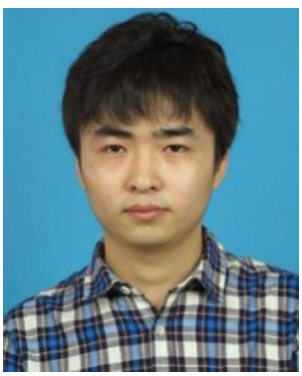
David, Y., & Yahav, E. (2014). Tracelet-based code search in executables. In *ACM SIGPLAN Conference on programming language design & implementation* (Vol. 49, pp. 349–360).
Dd-wrt r21676 (2013). Available at http://tinyurl.com/ddwrt-21676. Accessed 3 March 2017.
Dd-wrt. r21676 (2013). Available at http://tinyurl.com/ddwrt-21676. Accessed May 2013.
Diestel, R. (1997). Graph theory. *Mathematical Gazette*, *173*(502), 67–128.
Dir-600 firmware image (2013). Available at ftp://ftp2.dlink.com/PRODUCTS/DIR-600/. Accessed 3 March 2017.
Dir-645 firmware image (2013). Available at ftp://ftp2.dlink.com/PRODUCTS/DIR-645/. Accessed 3 March 2017.
Dir-815 firmware image (2014). Available at ftp://ftp2.dlink.com/PRODUCTS/DIR-815/. Accessed 3 March 2017.
Dir-300 firmware image (2014). Available at ftp://ftp2.dlink.com/PRODUCTS/DIR-300/. Accessed 3 March 2017.
Egele, M., Woo, M., Chapman, P., Brumley, D. (2014). Blanket execution: dynamic similarity testing for program binaries and components. In *USENIX conference on security symposium* (pp. 303–307).
Eschweiler, S., Yakdan, K., Gerhards-Padilla, E. (2016). discovre: Efficient cross-architecture identification of bugs in binary code. In: *Network and distributed system security symposium* (pp. 381–396).
Feldt, A., & Magazinius, R. (2010). Validity threats in empirical software engineering research - an initial survey. In *International conference on software engineering & knowledge engineering* (pp. 374–379).
Feng, Q., Zhou, R., Cheng, Y.C., Testa, B., Yin, H. (2016). Scalable graph-based bug search for firmware images. In *ACM SIGSAC Conference on computer and communications security* (pp. 480–491). ACM.
Gao, D., Reiter, M., Song, D. (2008). Binhunt: automatically finding semantic differences in binary programs. In *Information and communications security* (pp. 238–255).
Guifanow, I. (2016). Fast library identification and recognition technology in ida pro, available: http://www.hex-rays.com/idapro/. Accessed 5 Dec 2016.
Hsu, C.C., Yang, C.Y., Yang, J.S. (2005). Associating kNN and SVM for higher classification accuracy. In *Computational intelligence and security* (pp. 550–555).
Jang, J., Agrawal, A., Brumley, D. (2012). Redebug: finding unpatched code clones in entire os distributions. In *Security and privacy (SP)* (pp. 48–62). IEEE.
Jiang, L., Misherghi, G., Su, Z., Glondu, S. (2007). Deckard: scalable and accurate tree-based detection of code clones. In *International conference on software engineering* (pp. 96–105).
Jin, W., Chaki, S., Cohen, C., Gurfinkel, A., Havrilla, J., Hines, C. (2012). Binary function clustering using semantic hashes. In *International conference on machine learning and applications* (pp. 386–391). IEEE.
Kamiya, T., Kusumoto, S., Inoue, K. (2002). Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, *28*(7), 654–670.
Knownsec (2017). A statistical analysis report about the back door of d-link by zoomeye.org.knownsec.com [eb/ol]. Accessed 5 Sept 2017.
Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C. (2012). Rozzle: de-cloaking internet malware. In *Security and privacy (SP)* (pp. 443–457). IEEE.
Kolter, J.Z., & Maloof, M.A. (2004). Learning to detect malicious executables in the wild. In *International conference on knowledge discovery and data mining* (pp. 470–478).
Lakhotia, A., Preda, M.D., Giacobazzi, R. (2013). Fast location of similar code fragments using semantic 'juice'. In *ACM SIGPLAN Program protection and reverse engineering workshop* (pp. 1–6). ACM.
Li, Z., Lu, S., Myagmar, S., Zhou, Y. (2004). Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Symposium on opearting systems design & implementation* (Vol. 4, pp. 289–302).
Lin, H., Zhao, D., Ran, L., Han, M., Tian, J., Xiang, J. (2017). Cvssa: cross-architecture vulnerability search in firmware based on support vector machine and attributed control flow graph. In *The fourth international conference on dependable systems and theirs applications (DSA 2017)* (pp. 35–41).
Ming, J., Pan, M., Gao D. (2012). ibinhunt: binary hunting with inter-procedural control flow. In *International conference on information security and cryptology* (pp. 92–109). Berlin: Springer.
Myles, G., & Collberg, C. (2005). K-gram based software birthmarks. In *ACM Symposium on applied computing* (pp. 314–318).
Netgear readynas v6.1.6 (2014). Accessed 3 March 2017.
Ng, B.H., & Prakash, A. (2013). Expose: discovering potential binary code re-use. In *Computer software and applications conference* (pp. 492–501). IEEE.
Open web application secutity project (2016). https://www.owasp.org/index.php/owasp_internet_of_things_top_ten_project#tab=top_10_iot_vulnerabilities_282014_29. Accessed 5 Feb 2016.

Openssl v1.0.1.e (2013). https://www.openssl.org/source/old/1.0.1/openssl-1.0.1e.tar.gz. Accessed 3 March 2017.
Openssl v1.0.1f (2014). https://www.openssl.org/source/old/1.0.1/openssl-1.2.1.tar.gz. Accessed 5 Jan 2017.
Pewny, J., Schuster, F., Bernhard, L., Holz, T., Rossow, C. (2014). Leveraging semantic signatures for bug search in binary programs. In *Computer security applications conference* (pp. 406–415). ACM.
Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T. (2015). Malicious code detection for open firmware. In *Security and privacy (SP)* (pp. 709–724). IEEE.
Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Vigna, G. (2016). Driller: augmenting fuzzing through selective symbolic execution. In *Network and distributed system security symposium* (pp. 1–16).
Tabrizi, S.S., & Cavus, N. (2016). A hybrid kNN-SVM model for iranian license plate recognition. *Procedia Computer Science*, *102*, 588–594.
Ukil, A. (2002). Support vector machine. *Computer Science*, *1*(4), 1–28.
Yamaguchi, F., Golde, N., Arp, D., Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *Security and privacy (SP)* (pp. 590–604). IEEE.
Yusof, A.R.A., Udzir, N.I., Selamat, A. (2016). An evaluation on kNN-SVM algorithm for detection and prediction of DDoS attack. In *International conference on industrial, engineering and other applications of applied intelligent systems* (pp. 550–555). Springer International Publishing.
Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D. (2014). Avatar: a framework to support dynamic security analysis of embedded systems' firmwares. In *Network and distributed system security symposium* (pp. 1–16).
Zhang, H., Berg, A.C., Maire, M., Malik, J. (2006). SVM-kNN: discriminative nearest neighbor classification for visual category recognition. *CVPR2006*, *2*, 2126–2136.

Dongdong Zhao received Ph.D. degree from the University of Science and Technology of China (USTC) in 2016. He is currently a lecturer of the School of Computer Science and Technology of Wuhan University of Technology. His research interests include dependable computing, information security, privacy protection, and biometrics.

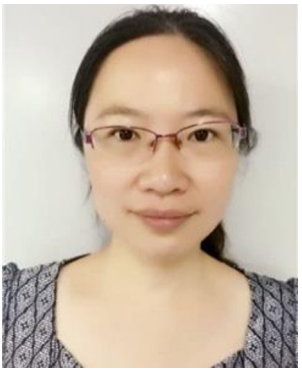
Hong Lin is working towards the master's degree in Wuhan University of Technology. Her research interests include information security and vulnerability search.

Linjun Ran is working towards the master's degree in Wuhan University of Technology. His research interests include information security, IoT security, and vulnerability detection of firmware.



Mushuai Han is working towards the master's degree in Wuhan University of Technology. His research interests include information security and vulnerability search.

Jing Tian received PhD degree in Knowledge Science from Japan Advanced Institute of Science and Technology (JAIST) in September 2006. She worked as postdoctoral researcher at JAIST and Japan National Institute for Environment Studies from October 2006 to February 2008. She is currently an associate professor of the School of Computer Science and Technology at Wuhan University of Technology, China. Her research interests are knowledge management, system science, and software engineering.



Liping Lu is currently an associate professor at the College of Computer Science and Technology, Wuhan University of Technology. Her research interests include Internet of Vehicles, network simulation, and embedded system.

Shengwu Xiong received the B.Sc. degree in computational mathematics from Wuhan University, Wuhan, China, in 1987, and the M.Sc. and Ph.D. degrees in computer software and theory from Wuhan University, in 1997 and 2003, respectively. Currently, he is a professor with the School of Computer Science and Technology, Wuhan University of Technology, China. His research interests include intelligent computing, machine learning, and pattern recognition.

Jianwen Xiang received Ph.D. degrees from Wuhan University and from Japan Advanced Institute of Science and Technology (JAIST) in 2004 and 2005, respectively. He is currently a Professor of the School of Computer Science and Technology of Wuhan University of Technology, and he was a researcher at NEC Corporation from 2008 to 2014. His research interests include dependable computing, information security, and software engineering.