Scalable Graph-based Bug Search for Firmware Images

Qian Feng[†], Rundong Zhou[†], Chengcheng Xu[†], Yao Cheng[†], Brian Testa[†◊], and Heng Yin[†*]

[†]Department of EECS, Syracuse University, USA
[◊]Air Force Research Lab
[*]University of California, Riverside
[†]{qifeng,rzhou02,cxu100,ycheng,heyin}@syr.edu [◊]brian.testa.1@us.af.mil [*]heng@cs.ucr.edu

ABSTRACT

Because of rampant security breaches in IoT devices, searching vulnerabilities in massive IoT ecosystems is more crucial than ever. Recent studies have demonstrated that control-flow graph (CFG) based bug search techniques can be effective and accurate in IoT devices across different architectures. However, these CFG-based bug search approaches are far from being scalable to handle an enormous amount of IoT devices in the wild, due to their expensive graph matching overhead. Inspired by rich experience in image and video search, we propose a new bug search scheme which addresses the scalability challenge in existing cross-platform bug search techniques and further improves search accuracy. Unlike existing techniques that directly conduct searches based upon raw features (CFGs) from the binary code, we convert the CFGs into high-level numeric feature vectors. Compared with the CFG feature, high-level numeric feature vectors are more robust to code variation across different architectures, and can easily achieve real-time search by using state-of-the-art hashing techniques.

We have implemented a bug search engine, `Genius`, and compared it with state-of-art bug search approaches. Experimental results show that `Genius` outperforms baseline approaches for various query loads in terms of speed and accuracy. We also evaluated `Genius` on a real-world dataset of 33,045 devices which was collected from public sources and our system. The experiment showed that `Genius` can finish a search within 1 second on average when performed over 8,126 firmware images of 420,558,702 functions. By only looking at the top 50 candidates in the search result, we found 38 potentially vulnerable firmware images across 5 vendors, and confirmed 23 of them by our manual analysis. We also found that it took only 0.1 seconds on average to finish searching for all 154 vulnerabilities in two latest commercial firmware images from D-LINK. 103 of them are potentially vulnerable in these images, and 16 of them were confirmed.

Keywords

Firmware Security; Machine Learning; Graph Encoding

CCS'16, October 24-28, 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978370

1. INTRODUCTION

Finding vulnerabilities in devices from the Internet of Things (IoT) ecosystem is more crucial than ever. Unlike in PCs or mobile phones, a security breach in one IoT device could cause unprecedented damage to our daily life, involving massive breakdowns of public systems [4] or quality of life issues. Even worse, we cannot rely on traditional protection mechanisms like commercial AntiVirus software on PCs and mobile devices to prevent attacks, since these traditional defenses are not feasible on IoT devices due to their relatively low CPU and memory profiles [3]. Gartner, Inc. forecasts that 6.4 billion connected things will be in use worldwide in 2016, up 30 percent from 2015, and will reach 20.8 billion by 2020. The vast diffusion of devices will increase the potential for the introduction of vulnerabilities to the IoT ecosystem. As a result, the need for third-party evaluators (e.g. consumer product evaluators, penetration testers) to quickly and accurately identify vulnerabilities in IoT ecosystem devices on behalf of customers and the need to support periodic security evaluations on existing devices is increasing dramatically [1].

However, discovering vulnerabilities in an IoT ecosystem is like finding a needle in a haystack, even when we are dealing with known vulnerabilities. For many IoT products, security is an afterthought. Between copy-paste coding practices and outsourcing of functionality to untrusted third-party libraries, the development process of IoT devices is a fertile environment for bug generation and persistence. As several integration vendors may rely upon the same subcontractors, tools, or SDKs provided by third-party vendors [19], bugs generated during the development process can be spread across hundreds or even thousands of IoT devices with similar firmware. Without detailed knowledge of the internal relationships between these vendors, it is impossible to track the same vulnerability across the IoT ecosystem [19, 27]. It is even worse when these devices are built on different architectures [45].

To address this critical problem, security researchers have been actively developing techniques to automatically analyze and detect vulnerabilities in IoT products [17,52,61]. Advancing our ability to perform bug searches in a general, lightweight manner is becoming increasingly important. Such a bug search technique would allow security professionals to reduce the time and resources required to locate a problem. Having done so, security professionals could scan all IoT devices in the ecosystem for a new vulnerability, and quickly generate a security evaluation. They would also be able to scan a new device for all known vulnerabilities, thus allowing us to learn from past mistakes before the possibly bug-infested IoT devices are widely deployed.

Bug search at scale. Researchers have already made primitive attempts at bug searching in the firmware images of IoT devices [19, 23,45]. One common approach is to scan the firmware images using superficial patterns such as constant numbers or specific strings [19]. When a vulnerability is tied to a unique constant or string, this approach can be very effective and can easily scale to a large number of firmware images. However, this approach lacks the generality to deal with more complex vulnerabilities that lack these distinct constant or string patterns.

To address cross-platform bug search in general, two recent efforts [23, 45] proposed extraction of various robust features from binary code and then performed searches against the extracted control flow graphs (CFGs). However, these approaches are far from being scalable. The approach by Pewny et al. [45] can take up to one CPU month to prepare and conduct a search in a stock Android image with 1.4 million basic blocks [23]. Eschweiler et al. [23] attempted to address this issue by leveraging a more lightweight feature extraction and using a pre-filtering technique. However, the efficacy of pre-filtering was only evaluated on a small number of firmware images, for a small set of vulnerabilities. According to our large-scale evaluation in Section 5.3, pre-filtering seems unreliable and can cause significant degradation in search accuracy.

Bug search as a search problem. Fundamentally, the bottleneck for the CFG-based bug search techniques is not about the graph matching algorithm, but rather the search scheme. These techniques conduct pairwise graph matching for search, the complexity of which makes them unusable in large-scale datasets.

A similar problem has been extensively studied in the computer vision community where they are interested in efficiently searching for similar images from large volume of image [30, 32, 60]. Instead of conducting pair-wise matching on raw images directly, they learn higher-level numeric feature representations from raw images. In this paper, we leverage the successful techniques from the computer vision community and propose a novel search method called Genius[1] for bug search in IoT devices. Although images appear to be quite different from binary functions, we found in this paper that scalable image search methodologies can be applied for graph-based bug search. As opposed to directly matching two control-flow graphs, the proposed method learns higher-level numeric feature representations from control-flow graphs, and then conducts search based on the learned higher-level features.

Compared with existing bug search methods, Genius has the following two benefits: First, the learned features tend to be more invariant to cross-architecture changes than the raw CFG features. This is because Genius learns a feature representation using representative CFGs, which tends to be more invariant than the pairwise match. This invariance property has been verified in a number of image retrieval system [53,59] such as the Google image search. The experimental results in Section 5 substantiate this hypothesis in the bug search problem. Second, Genius significantly improves the state-of-the-art bug search efficiency. The graph is a complex structure, and thus directly indexing or hashing graphs at scale is still a challenging problem [36]. However, the proposed method circumvents the difficulty by transforming a graph as a higher-level feature representation, which can be indexed by Locality Sensitive Hashing [54]. This transformation enables a bug search method that could be orders of magnitude faster than existing methods.

Our interdisciplinary study focuses on leveraging existing image retrieval techniques to address the scalability issue in the existing graph-based bug search methods. Furthermore, our empirical studies provide compelling insights how to find appropriate settings for the bug search problem. Our empirical observations may benefit not only the bug search methods but also other related approaches [24, 62, 63].

We have implemented a proof-of-concept Genius, and compared it with existing state-of-the-art bug search approaches; our experiments demonstrate that Genius outperforms existing methods over various query loads in terms of accuracy and efficiency and scalability. We further demonstrate the efficacy of IoT bug search on 8,126 firmware images of 420,558,702 functions. The performance testing on 10,000 queries showed that Genius can finish a query in less than 1 second on average.

Contributions. In summary, our contributions are as follows:

- **New insights.** We leverage techniques widely used in the computer vision community and develop a control flow graph based method to address the scalability issue in existing bug search techniques. We systematically study different schemes in the existing image search techniques, and discover an appropriate scheme for the bug search in IoT ecosystem.
- **Significant scalability improvement.** We demonstrate the efficacy of Genius on firmware images from 8,126 devices across three architectures and 26 vendors. The performance testing on 10,000 queries shows that Genius can process a query in less than 1 seconds on average. We also demonstrate that Genius can achieve comparable or even better accuracy and efficiency than the baseline techniques.
- **New Discoveries.** We demonstrate two use scenarios for Genius. The results show that Genius takes less than 3 seconds on average to finish a vulnerability search across 8,126 devices. As a result of the efficient method, by only looking at top 50 candidates in the search result, we found 38 potentially vulnerable firmware images across 5 vendors, and confirmed 23 of them via manual analysis. We also found that Genius takes only 0.1 second on average for all 154 vulnerabilities in two latest commercial firmware images from D-LINK. 103 potential vulnerabilities were found in these images, and 16 of them were confirmed.

2. APPROACH OVERVIEW

Inspired by image retrieval techniques, the proposed method includes the following main steps, as shown in Figure 1: 1) *raw feature extraction*, 2) *codebook generation*, 3) *feature encoding*, and 4) *online search*. The first step aims at extracting the attributed control flow graph, which is referred to as the raw feature, from a binary function (Section 3). *Codebook generation* utilizes unsupervised learning methods to learn higher-level categorizations from raw attributed control flow graphs (Section 4.1). *Feature encoding* encodes the attributed control flow graph by learned categorizations into higher-level feature vector residing in the high-dimensional space (Section 4.2). Finally, given a function, *online search* aims at efficiently finding its most similar functions by Locality Sensitive Hashing (LSH) [9]. Since each function is transformed into a higher-level numeric feature in the *feature encoding* step, we can directly apply LSH to conduct efficient searches in terms of the approximated cosine and Euclidean distance between the higher-level features (Section 4.3). The details of each step will be discussed in the following sections.

Generally, there are two stages in the proposed method: offline indexing and online search. Offline indexing, which includes raw feature extraction, codebook generation and feature encoding, is applied to existing functions before we can perform searches. Similar to text and image search methods, this step is a one-time effort and can be trivially paralleled across multiple CPU cores. The online search phase, which includes feature encoding and search, is

[1] Genius stands for Graph Encoding for Bug Search.

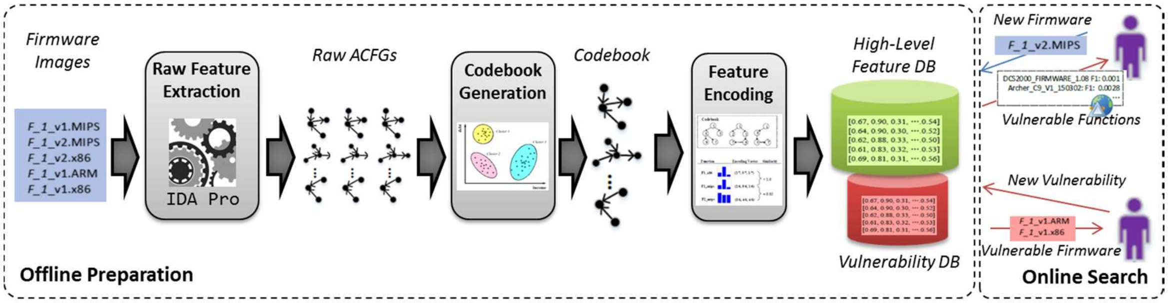


Figure 1: **The approach overview**

applied against a few unseen functions. Due to the limited number of online operations, online search is typically sufficiently fast for large-scale search engines. We elaborate on the two stages and the main steps in the following two use scenarios.

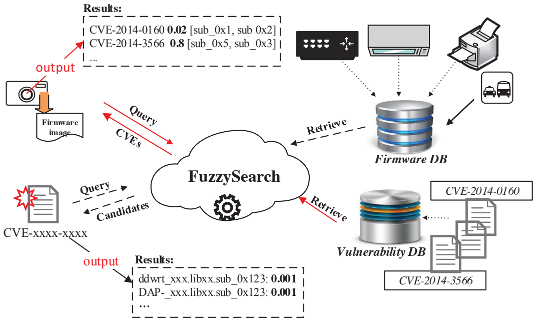


Figure 2: **The deployment of** Genius**.**

Deployment: Figure 2 shows two use scenarios for Genius. 1) *Scenario I*: Given a device repository, Genius will index functions in the firmware images of all devices in the repository based upon their CFGs. When a new vulnerability is released, a security professional can use Genius to search for this vulnerability in their device repositories . Genius will generate the query for the vulnerability and query in the indexed repository. The outputs will be a set of metadata about all potentially infected devices with their brand names, library names and the potentially vulnerable functions. All outputs will be ranked by their similarity scores for quick screening of the results. 2) *Scenario II*: Security professionals may upload unseen firmware images that do not exist in the repository for a comprehensive vulnerability scan. In this case, Genius will index these firmware images for the security professionals. As a result, they can simply query any vulnerabilities in our vulnerability database. Genius will retrieve the most similar vulnerabilities in the existing indexed firmware images and output metadata of all potentially vulnerable functions including their names, library names holding these functions and firmware device types where these functions are used. Again, all outputs will be ranked by their similarity scores to facilitate quick screening of the results.

3. RAW FEATURE EXTRACTION

The CFG (Control Flow Graph) is the common feature used in bug search. More recently, different attributes on the basic blocks, such as I/O pairs and statistic features [23, 45], are explored to increase the matching accuracy. Following the idea, this paper utilizes the control flow graph with different basic-block level attributes called ACFG (Attributed Control Flow Graph) as the raw features to model the function in our problem.

Definition 1. *(Attributed Control Flow Graph) The attributed control flow graph, or ACFG in short, is a directed graph $G = \langle V, E, \phi \rangle$, where V is a set of basic blocks; $E \subseteq V \times V$ is a set of edges representing the connections between these basic blocks, and $\phi : V \to \Sigma$ is the labeling function which maps a basic block in V to a set of attributes in Σ.*

The attribute set Σ in Definition 1 can be tailored depending upon the level of detail required to accurately characterize a basic block. For efficiency, instead of using expensive semantic features like I/O pairs [45], we focus on two types of features in this paper: statistical and structural. The statistical features describe local statistics within a basic block, whereas the structural features capture the position characteristics of a basic block within a CFG. Inspired by [23], in Σ we extract six types of statistical features and two types of structural features, listed in Table 1.

Inspired by the work on complex network analysis, we propose two types of structural features: *no. of offspring* and *betweenness centrality*. The *no. of offspring* is the number of children nodes in the control flow graph. This information helps locate the layer of a node in the graph. The *betweenness centrality* measures a node's centrality in a graph [42]. Nodes in the same layer in the CFG could have different betweenness centrality. The results in Section 5.4 demonstrate the efficacy of the proposed structural features. In summary, the proposed features consider not only the statistical similarity but also the structural similarity between two ACFGs.

To generate the attributed graph for a binary function, we first extract its control flow graph, along with attributes in Σ for each basic block in the graph, and store them as the features associated with the basic block. We utilize IDA Pro [28], a commercial disassembler tool, for attributed graph construction.

Table 1: Basic-block level features used in Genius.

Type	Feature Name	Weight (α)
Statistical Features	String Constants	10.82
	Numeric Constants	14.47
	No. of Transfer Instructions	6.54
	No. of Calls	66.22
	No. of Instructions	41.37
	No. of Arithmetic Instructions	55.65
Structural Features	No. of offspring	198.67
	Betweeness	30.66

4. METHODOLOGY

Section 3 introduced the notion of Attributed Control Flow Graph (ACFG), which will be used as the raw feature set for Genius. This section discusses how we utilize these ACFGs and transform them into high-level feature vectors that are suitable for scalable, accurate bug search.

4.1 Codebook Generation

The first step in the proposed method is codebook generation, which aims at learning a set of categorizations, that is codewords, from raw features. Formally, a codebook $\mathcal{C}$ is a finite and discrete set: $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$, where c_i is the i-th codeword, or "centroid", and i is the integer index associated with that centroid. The codebook is generated from a training set of raw features by an unsupervised learning algorithm. In our case, the raw features are the control flow graphs. The codebook generation consists of two phases: similarity metric computation and clustering.

4.1.1 Raw Feature Similarity

We consider the raw feature similarity computation as a labeled graph matching problem. By definition, ACFGs are matched not only by their structures but also by their labels (attributes) on the structures. Theoretically, graph matching is an NP-complete problem, but many techniques have been proposed to optimize the process for an approximate matching result [14,49]. For efficiency, we utilize bipartite graph matching to quantify ACFG similarity. Although other approaches such as MCS (Maximum Common Subgraph) matching [14] may also be applied to this problem, efficiency is still a major concern. The primary limitation of bipartite graph matching is that it is agnostic to the graph structure, and the accumulation of errors could result in less accurate results. To address the issue, we have appended structural features, described in Section 3, to allow bipartite graph matching to incorporate some graph structural information. Experiments in Section 5.4 show that these structural features boost the accuracy of bipartite graph matching in our problem.

Essentially, bipartite graph matching utilizes the match cost of two graphs to compute the similarity. It quantifies the match cost of two graphs by modeling it as an optimization process. Given two ACFGs, G_1 and G_2, the bipartite graph matching will combine two ACFGs as a bipartite graph $G_{bp} = (\hat{V}, \hat{E})$, where $\hat{V} = V(G_1) \cup V(G_2)$, $\hat{E} = \{\hat{e_k} = (v_i, v_j) | v_i \in V(G_1) \wedge v_j \in V(G_2)\}$, and edge $\hat{e_k} = (v_i, v_j)$ indicates a match from v_1 to v_2. Each match is associated with a cost. The minimum cost of two graphs is the sum of all edges cost on the mapping. Bipartite graph matching can go over all mappings efficiently, and select the one-to-one mapping on nodes from G_1 to G_2 of the minimum cost.

In our problem, a node in the bipartite graph is a basic block on the ACFG. The edge cost is calculated by the distance between the two basic blocks on that edge. Each basic block on the ACFG has a feature vector discussed in Section 3. Therefore, we calculate the distance between two basic blocks by $\text{cost}(v, \hat{v}) = \frac{\sum_i \alpha_i |a_i - \hat{a}_i|}{\sum_i \alpha_i \max(a_i, \hat{a}_i)}$. It is the same distance metric used in the paper [23] to quantify the distance of two basic blocks. a_i and $\hat{a_i}$ are the i-th feature in feature vectors of two basic block v and $\hat{v}$ respectively. If the feature is a constant, $|a_i - \hat{a_i}|$ is their difference. If the feature is a set, we use Jaccard to calculate the set difference. α_i is the corresponding weight of the feature which will be discussed below.

The output of bipartite graph matching is the minimum cost of two graphs. Normally the match cost of two graphs is greater than one, and positively correlated to the size of compared ACFGs. Therefore, we normalize the cost to compute the similarity score. For cost normalization, we create an empty ACFG Φ for each compared ACFG. Each node in the empty graph has an empty feature vector, and the size of the empty graph is set to that of the corresponding compared graph. By comparing with this empty ACFG, we can obtain the maximum matching cost the compared graph can produce. We compute the matching cost with the empty graph for the two graphs, and select the maximum matching cost as the denominator, and use it to normalize the matching cost of two graphs. Suppose $cost(g_i, g_j)$ represents the cost of the best bipartite matching between two graphs g_1, g_2, the ACFG similarity between two graphs can be formally represented as following:

$$\kappa(g_1, g_2) = 1 - \frac{\text{cost}(g_1, g_2)}{\max(\text{cost}(g_1, \Phi), \text{cost}(\Phi, g_2))}, \qquad (1)$$

We found that the features in Table 1 have different importance in computing graph similarity. We learn weights of the raw features to capture the latent similarity between two ACFGs. Basically, the learning objective is to find weight parameters that can maximize the distance of different ACFGs while simultaneously minimizing the distance of equivalent ACFGs. To approach this optimization problem, we adopt the approach used in Eschweile et al [23]. More specifically, we use a genetic algorithm using GALib [56]. We also execute an arithmetic crossover using a Gaussian mutator 100 times. The learned weights for each feature are listed in Table 1.

4.1.2 Clustering

After defining the similarity metric for the ACFG, the next step is to generate a codebook using the unsupervised learning method. This process can be regarded as a clustering process over a collection of raw features: ACFGs, where each cluster comprises a number of similar ACFGs.

In this paper, we use spectral clustering [43] as the unsupervised learning algorithm to generate the codebook. Formally, the spectral clustering algorithm partitions the training set of ACFGs into n sets $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ so as to minimize the sum of the distance of every ACFG to its cluster center. $c_i \in \mathcal{C}$ is the centroid for the subset S_i. We define the centroid node as the ACFG that has the minimum distance to all the other ACFGs in S_i, and the collection of all centroid nodes constitutes a code book.

Unlike traditional clustering algorithms, in which the inputs are numerical vectors, in this paper we propose to use a kernelized spectral clustering where the input is a kernel matrix. The similarity computed in Section 3 can be used to generate the kernel matrix for the spectral clustering. Suppose the kernel matrix is $\mathbf{M}$, and each entry in $\mathbf{M}$ is a similarity score of two corresponding ACFGs. The kernelized clustering works on $\mathbf{M}$ and outputs the optimal partitions (clusters) of ACFGs in the training data.

The codebook size n would affect the bug search accuracy. To this end, we systematically study a suitable n in the bug search in Section 5 and demonstrate that $n = 16$ seems to be a reasonable codebook size trading off efficiency and accuracy.

In order to reduce computational cost in constructing the codebook, a common strategy is to randomly sample a training set from the entire dataset. We observed that there is a significant variance in ACFG size. To reduce the sampling bias, we first collect a dataset which covers ACFGs of different functions from various architectures. See Section 5.2. Then split ACFGs into separate "strata" with different size ranges. Each stratum is then sampled as an independent sub-population, out of which individual ACFGs are randomly selected. This is a commonly used approach known as stratified sampling [50].

The codebook generation is expensive. However, since the codebook generation is an offline and one-time effort, it will not detrimentally impact the runtime for the online searches. Besides, some approaches can be used to expedite this process, such as the parallelled clustering approximate clustering [12] or the hierarchical clustering algorithm [40].

4.2 Feature Encoding

Given a learned codebook, feature encoding is to map raw features of a function into a higher-level numeric vector, each dimension of which is the similarity distance to a categorization in the codebook. This step is known as feature encoding [16].

There are two notable benefits for feature encoding. First, the higher-level feature can better tolerate the variation of a function across different architectures, as each of its dimensions is the similarity relationship to a categorization which is less sensitive to the variation of a binary function than the ACFG itself. This property has been demonstrated by many practices in the image search to reduce the noises from the scale, viewpoint and lighting. We further demonstrate it in the bug search scenario in Section 5.3. Second, the ACFG raw features after encoding becomes a point in the high dimensional space which can be conveniently indexed and searched by existing hashing methods. Therefore, the encoding enables a faster real-time bug-search system. See Section 5.5.

Formally, the feature encoding is to learn a quantizer $q : \mathbb{G} \rightarrow \mathbb{R}^n$ over the codebook $\mathcal{C} = \{c_1, ..., c_n\}$, where $\mathbb{G}$ is the set of all ACFG graphs following Definition 1, and $\mathbb{R}^n$ represents the n-dimensional real space. In this paper, we discuss two approaches to derive q. For a given graph g_i, let $NN(g_i)$ represent the nearest centroid neighbors in the codebook:

$$NN(g_i) = \arg\max_{c_j \in \mathcal{C}} \kappa(g_i, c_j) \tag{2}$$

where κ is defined in Eq. (3). A common practice in image retrieval is to consider not only the nearest neighbor but a few nearest neighbors, e.g. 10 nearest neighbors [31, 59].

Bag-of-feature encoding. The bag-of-feature encoding, which maps a graph to some centroids in the codebook, represents each function as a bag of features. The bag-of-feature quantizer can be defined as:

$$q(g_i) = \sum_{g_i : NN(g_i) = c_j} [\mathbb{1}(1 = j), \ldots, \mathbb{1}(n = j)]^T, \tag{3}$$

where $\mathbb{1}(\cdot)$ is an indicator function which equals 1 when $\cdot$ is true and 0 otherwise. Eq. (3) indicates that the output encoded feature will add 1 to the corresponding dimension of the nearest centroid. This representation is inspired by the bag-of-words model used in text retrieval [38], where each document is represented by a collection of terms in the English vocabulary. In analogy, in our problem, each function is represented by a collection of representative graphs in the learned codebook. After encoding each function becomes a point in the high dimensional vector space.

VLAD encoding. The drawback of the bag-of-word model is that the distance between a given graph and a centroid is completely ignored as long as the centroid is the graph's nearest neighbors. The VALD [10] encoding was proposed to incorporate the first-order differences and assigns a graph to a single mixture component.

$$q(g_i) = \sum_{g_i : NN(g_i) = c_j} [\mathbb{1}(1 = j)\kappa(g_i, c_1), ..., \mathbb{1}(n = j)\kappa(g_i, c_n)]^T, \tag{4}$$

Compared to Eq. (3), Eq. (4) adds the similarity information to the centroids in the encoded features. Note as our raw features are graphs, in Eq. (4) we use the kernelized similarity function in the VALD encoding which is different from the traditional VALD defined for image retrieval. In VALD encoding, a dimension represents the similarity to a corresponding ACFG centroid in the codebook. As a result, the vector is of latent semantic meaning that reflects a similarity distribution across all centroids in the learned codebook. Empirically we found that VLAD encoding performs better than the bag-of-feature encoding for bug search.

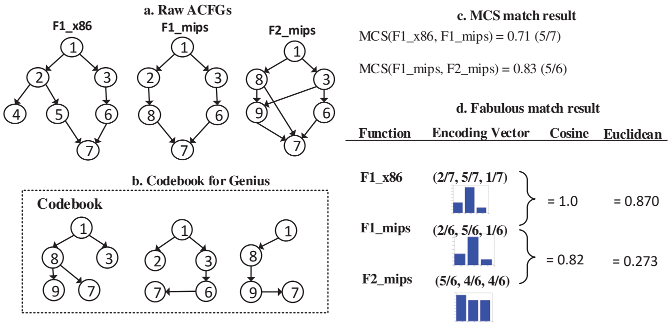


Figure 3: A toy example on VALD feature encoding. Features on each basic block of ACFG are simplified into a single constant value. The match cost of two basic blocks with the same value is zero, otherwise 1.

We will walk the encoding algorithm in a toy example in Fig. 3. Given ACFGs for three functions F1_x86, F1_mips and F2_mips, where the first two are the same function compiled from different architectures, and the last one is a completely different function. The compiler used for F1_mips merges the basic block 4 and 5 into the single node 8, so the ACFG of function F1 in MIPS is different from that in x86, due to the instruction reordering. F2_mips shares partial code with function F1_mips. For example, they both check some environment conditions and directly return if it fails.

For clarity, the similarity metric between ACFGs used in Fig. 3 adopts maximum common subgraph matching. For example, the similarity score between F1_x86 and F1_mips is $5/7$ where 5 means the maximum common subgraph, and 7 means the maximum size between two graphs.

The pair-wise match will match two functions directly by their ACFGs, whereas `Genius` will match them by their encoded vectors. VLAD encoding generates the encoded vector by comparing a ACFG to its top 3 closest centroid nodes in the codebook in Fig. 3b). Different from BOF model, it will store the similarity score to each centroid node into the corresponding dimension in the vector. The resulting feature vector is shown in Fig. 3d) (bottom-right corner). Fig. 3d) also lists match results for both pair-wise match and `Genius`. The distance metrics used by `Genius` will be discussed in Section 4.3. We can see that the pair-wise graph match fails to match F1_x86 to F1_mips, since it matches two functions locally. On the contrary, `Genius` can still match these two functions with high similarity score, as the encoded feature vector is more invariant to local changes on an ACFG. Note, this toy example is only for illustration and we will substantiate our hypothesis by extensive experiments on real-world datasets in Section 5.3.

4.3 Online Search

The encoded features generated in Section 4.2 may be directly used in search. However, this straightforward solution may not be scalable for millions of functions in real-world applications. This section introduces a scalable solution by LSH (Locality-sensitive hashing) to scale the search. In this paper, we utilize LSH as opposed to other indexing methods such as k-d tree, as the k-d tree may not be suitable for our problem due to its inefficiency in high-dimensional spaces especially when the codebook is large [57].

Given a query function, we first derive its encoded feature by Eq. (3) and (4), then we are interested in finding the function in a large dataset that are closest to the query with a high probability. LSH achieves this goal by learning a projection so that if two points are closer together in the feature encoding space, they should remain close after the projection in the hashing space. Following [54], given the encoded feature $q(g)$, we employ the projection functions h_i defined as:

$$h_i(q(g)) = \lfloor(\mathbf{v} \cdot q(g) + b)/w\rfloor, \tag{5}$$

where w is the number of quantized bin, $\mathbf{v}$ is a randomly selected vector from a Gaussian distribution, and b is a random variable sampled from a uniform distribution between 0 and w. In addition, $\lfloor\cdot\rfloor$ is he floor operator. Essentially, a hashing function defines a hyper-plane to project the input encoded features. For any functions $q(g_i), q(g_j) \in \mathbb{R}^n$ that are close to each other in the encoding space, there is a high probability $P[h(q(g_i)) = h(q(g_j))] = p_1$ that they fall into the same bucket. Likewise, for any functions that are far apart, there is a low probability $p_2(p_2 < p_1)$ that they fall into the same bucket.

The locality sensitive hash of an encoded feature $q(g)$ as $lsh(g) = [h_1(q(g)), .., h_w(q(g))]$ where w is the number of hash functions. After LSH, a function is projected as a point in the hashing space. We experiment on two classical distance metrics defined in the hashing space: Euclidean distance and the cosine distance [47] in our bug search problem.

5. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate Genius with respect to accuracy, efficiency, and scalability. First, we briefly describe the experiment setup and the data sets used in our evaluation (Sections 5.1 and 5.2). Second, we conduct a systematic baseline comparison against the existing bug search methods in terms of the accuracy and efficiency in the cross-platform setting (Section 5.3). Third, we evaluate Genius on 33,045 firmware images and demonstrate its scalability (Section 5.5). Finally, we present two case studies that demonstrate how we would expect a user to employ Genius under realistic conditions (Section 5.6).

5.1 Experiment Setup

We wrote the plugin to the disassembler tool IDA Pro [28] for ACFG extraction. We implemented codebook generation, feature encoding in python, and adopted Nearpy [6] for LSH hashing and search. We utilized MongoDB [5] to store the firmware images and encoded features. Our experiments were conducted on a server with 65 GB memory, 24 cores at 2.8 GHz and 2 TB hard drives. All the evaluations were conducted based on four types of datasets: 1) baseline evaluation dataset; 2) two public firmware images; 3) 33,045 firmware images and 4) the vulnerability dataset.

5.2 Data preparation

***Dataset I* – Baseline evaluation.** This dataset was used for baseline comparison, and all functions in this dataset has known ground truth for metric validation. We prepared this dataset using BusyBox (v1.21 and v1.20), OpenSSL (v1.0.1f and v1.0.1a) and coreutils (v6.5 and v6.7). All programs were compiled for three different architectures (x86, ARM, MIPS; all 32 bit) using three compiler versions (gcc v4.6.2/v4.8.1 and clang v3.0). We also enabled four optimization levels (O0-O3) for each version of a compiler. These settings have been used in existing studies as well [45]. We kept the symbol names during compilation which allowed us to maintain ground truth for evaluations.

These different compilation combinations resulted in a dataset of over 568,134 functions. As several of the existing techniques could not scale to a dataset of this size, we randomly sampled 10,000 functions from this dataset as the baseline dataset, and used that for baseline evaluation. Each function in the baseline dataset has at least two instances: one for query and another for search. The remaining functions in Dataset I were used for codebook generation.

***Dataset II* – Public dataset.** Recent work such as Pewny et al [45] and Eschweiler et al [23] used the same public dataset based upon two publicly-available firmware images for baseline comparison [7, 8]. We also evaluated Genius using this dataset to provide for fair comparison with the state-of-the-art systems.

***Dataset III* – Firmware image dataset.** This dataset of 33,045 firmware images was collected from the wild. We used it to evaluate the scalability of Genius. The images in this dataset were collected from three sources: 9,000 firmware images from [17], the entire dataset from [19] and 500 randomly selected images from our own crawl of the DDWRT ftp site [2]. Of the total 33,045 firmware images collected, we successfully unpacked 8,126 images from 26 different vendors. The vendors include such as ATT, Verizon, Linksys, D-Link, Seiki, Polycom, TRENDnet. The product types from each vendor include IP cameras, routers, access points and third-party or open-source firmware.

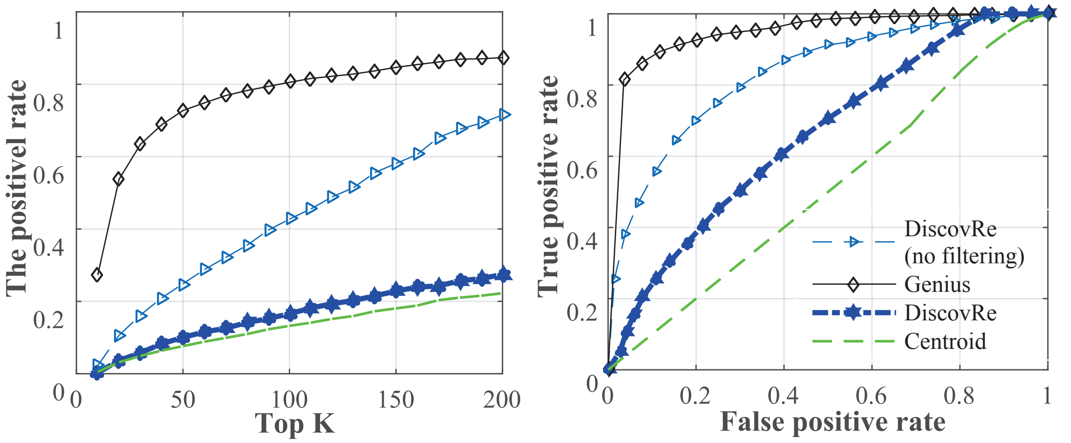
***Dataset IV*–The vulnerability dataset.** This dataset is a mapping between CVE numbers and the corresponding functions that actually introduced the vulnerabilities. To construct a query which can be used by Genius for bug search, we need to find binary code for these vulnerable functions. While other works have already investigated construction of vulnerability databases [44], none of them fit our purposes; they cannot extract the binary code for vulnerable functions required by Genius. As a result, we created a freely available vulnerability database for this effort and for the broader research community.

To build this database, we mined official software websites to collect lists of vulnerabilities with the corresponding CVE numbers. We were also able to retrieve information about the software commits to fix the vulnerabilities, which provided us the vulnerable function names and symbols. We then downloaded the source code for the vulnerable versions of the software, compiled the source and extracted the vulnerable functions from the binary code using the symbol names. We then used Genius to generate higher-level features for each vulnerable function. In the end, we utilized MongoDB [5] to build the database which stored the vulnerable functions and their corresponding feature vectors for later use. In our evaluation, we were only interested in vulnerabilities related to libraries widely used in firmware devices. We selected OpenSSL for demonstration, since it is widely used in IoT devices. The resulting vulnerability database includes 154 vulnerable functions.

5.3 Cross-Platform Baseline Comparison

We first evaluated Genius with baseline methods under the cross-platform setting. All evaluations in this subsection were conducted under the baseline dataset in Dataset I and Dataset II. Since each function name has multiple instances in Dataset I, we collected the query set Q by randomly selecting one instance for each function name, and considered the rest of the baseline dataset as a codebase. The codebook of Genius in this part is also trained on Dataset I, and its size is 16 for all baseline evaluations.

Evaluation Metrics. We used two metrics to evaluate the accuracy of the proposed and baseline methods: the recall rate (A.K.A true positive rate) and false positive rate. In the code search scenario, the search results are a ranked list. For each query q, there are m matching functions out of a total of L functions. If we consider the top-K retrieved instances as positives, the total number of correctly matched functions, μ, are true positives, and the remaining



a) Recall rates across different threshold K

b) ROC curves for different approaches

Figure 4: **Baseline comparison for accuracy on Dataset I.** K **means that we consider retrieved candidates on top K as positives** Two figures share the same legends.

number of functions in the top K, that is $K - \mu$, are false positives. Based on the definitions of recall and false positive rate, the recall rate is calculated as $recall(q) = \frac{\mu}{m}$ and the false positive rate is $FPR(q) = \frac{(K-\mu)}{L-m}$.

Preparation of Baseline Systems. We prepared three representative, state-of-the-art, cross-architecture bug search techniques to establish our evaluation baseline: discovRe [23], Multi-MH and Multi-k-MH [45] and a centroid based search [18]. Our first task was to prepare versions of these solutions for this evaluation.

- *discovRe* [23]: Due to unavailability of the source code, we reimplemented discovRe[2] and set the iteration limitation to be the same (i.e., 16*max($|G_1|, |G_2|$)) as the one used in their work work. We evaluated Genius against two versions of discovRe: the original version with pre-filtering and the version without pre-filtering. For the pre-filtering version, we set the threshold to 128 as specified in [23]. The version without pre-filtering uses only their graph matching metric for search.
- *Multi-MH and Multi-k-MH* [45]: Their source code is not available. Due to the complexity, it was less possible for us to reimplement their approach within a reasonable amount of time. Fortunately, discovRe has already conducted a thorough baseline comparison against these two approaches and published the results. The binaries used for the evaluation are also available online. Hence, we evaluated Genius on the same setting for this baseline comparison, and compared the published numbers on the benchmark.
- *Centroid* [18]: The centroid-based approach is known for its efficiency with respect to the Android malware clone and repackage problem [18]. We implemented a centroid-based bug search system for IOT devices. Note that while the centroid method is not directly designed for cross-platform code matching, it is still meaningful to compare it with Genius in terms of efficiency and accuracy.

A. Accuracy comparison.

To evaluate the efficacy of Genius, we first conducted thorough comparisons with DiscovRe and Centroid on Dataset I, since we have reimplemented these two approaches. We compared with the published results of Multi-MH and Multi-k-MH on Dataset II.

The first round of evaluations worked on the baseline dataset in Dataset I. We randomly selected 1000 functions as queries to feed into the target approach, and evaluated search results in terms of two metrics. Fig. 4a) lists the average recall rates for 1,000 queries across different thresholds of K, where the the y-axis indicates the recall for the corresponding K values. We can see that Genius significantly outperforms the baseline methods for every value of K. For example, Genius ranks 27% functions at top 1, whereas discovRe only ranks 0.5%. We also found that the performance of discovRe is worse than the version without pre-filtering. Fig. 4b) shows the ROC curves for each approach. This was the macro-average result across 1,000 queries. We can see that the ROC curve of Genius is better than those of the baseline approaches, especially when the false positive rate is small. The results in Fig. 4 (a) and (b) substantiate that Genius can achieve even better accuracy than the state-of-the-art methods.

We inspected the search results and found that the superior performance of Genius is mainly because of the salient and robust feature representation learned on top of the ACFGs. As an example, ssl3_get_message was ranked at top 1 by Genius but ranked below 40th by baseline methods because its CFG extracted from the function of X86 and MIPS was changed. Our method managed to capture the change and thus showed better results. We also analyzed the cases where Genius yield worse results than baseline methods. We hypothesize the reason is about the quality of the learned codebook. We will discuss it in Section 5.4.

B. Efficiency comparison.

We conducted efficiency comparison in terms of online search and offline preparation. For online search, we evaluated on both Dataset I and II. For offline preparation efficiency, we evaluated Dataset II as a demonstration.

Offline Preparation Efficiency. The preparation includes ACFG extraction and feature encoding. Table 3 shows the aggregation of preparation time for the phases of Genius on Dataset II. We can see that Genius outperforms Multi-MH and Multi-k-MH. DiscovRe only considers the control flow graph extraction time, whereas Genius needs extra time to encode these graphs. Even if Genius is slower than DiscovRe at the preparation stage, as the preparation

[2] We contacted the author of discovRe for comparison by providing their search results, but they have not provided us results yet.

Table 2: Comparison with Multi-MH and Multi-k-MH, discovRE, Centroid with the propose method for OpenSSL. Each cell contains the rank, separated by the colon, for both vulnerable functions: heartbeat for TLS and DTLS.

	Multi-MHTLS [45]		**Multi-k-MH** [45]		**discovRE** [23]		Genius		**Centroid** [18]	
From ->To	**TLS**	**DTLS**	**TLS**	**DTLS**	**TLS**	**DTLS**	**TLS**	**DTLS**	**TLS**	**DTLS**
MIPS → DD-WRT	1:2	2:4	1:2	1:2	1:2	1:2	1:2	1:2	46:100	87:99
MIPS → ReadyNAS	1:2	6:16	1:2	1:4	1:2	1:2	1:2	1:2	88:190	678:988
x86 → DD-WRT	70:78	1:2	5:33	1:2	1:2	1:2	1:2	1:2	97:255	102:89
x86 → ReadyNAS	1:2	1:2	1:2	1:2	1:2	1:2	1:2	1:2	145:238	333:127
Query Normalized Avg. Time	0.3s		1 s		4.1×10^{-4} s		1.8×10^{-6} s		1.4×10^{-6} s	

Table 3: Baseline comparison on preparation time.

			Preparation Time in Minutes				
Firmware Image	**Binaries**	**Basic Blocks**	**Multi-MH**	**Multi-k-MH**	**discovRE**	Genius	**Centroid**
DD-WRT r21676 (MIPS)	143 (142)	329,220	616	9,419	2.1	4.9	3.2
ReadyNAS v6.1.6 (ARM)	1,510 (1,463)	2,927,857	5,475	83,766	54.1	89.7	69.6

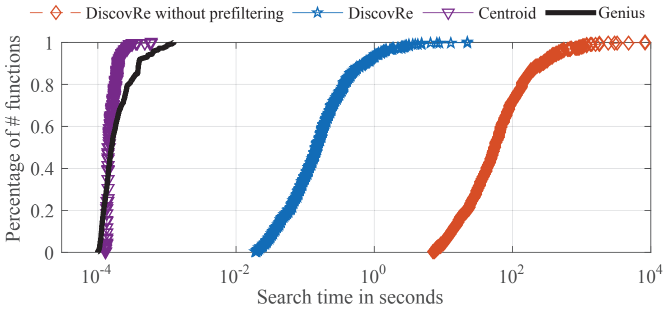


Figure 5: **The CDFs of search time on Dataset I.**

is an offline stage and only an one-time effort, it is reasonable to sacrifice some preparation time for the online search efficiency.

Online Search Efficiency. Similar to the accuracy comparison discussed above, we evaluated the online search efficiency on Dataset I and II, respectively. We first conducted the search on Dataset I, searched all of the functions in the dataset and recorded their search times for each target approach. Fig. 5 lists the Cumulative distribution function (CDFs) of search time for the four approaches on Dataset I, where the x-axis plots the search time in seconds. We can see that Genius and the centroid-based approach have least search time. DiscovRe, on the other hand, has the longest search time because it requires expensive online graph matching. In the best case, discovRe takes 10 ms for a query, whereas Genius only requires 0.1 ms to return more accurate results. Unsurprisingly, we also found that the version of discovRe without pre-filtering has even worse performance. It required nearly 2 hours for a single query in the worst case, and was still less accurate than Genius. Although the centroid approach had comparable efficiency with Genius, as previously mentioned, centroid significantly underperforms Genius in terms of the accuracy.

We also conducted a second round evaluation on Dataset II for all baseline approaches. We utilized the search time for the Heartbleed vulnerability as the metric. Table 2 lists the search results. It shows that Genius is orders of magnitude faster than Multi-MH, Multi-k-MH and discovRE. This demonstrates that Genius outperforms most of the existing methods in terms of efficiency.

5.4 Parameter Studies

We systematically studied the parameter's impact on the accuracy of Genius under different settings. The parameters for evaluation included the structural features used in bipartite graph matching, the codebook size, the size of training data for codebook generation, and the feature encoding methods. All evaluation settings were conducted on Dataset I.

A. Distance metrics and structural features. To verify the contribution of the proposed structural features, we conducted bipartite graph matching experiments with and without structural features. As shown in Fig. 6a), the matching with structural features outperforms the matching without it. Besides, we also evaluated two distance metrics used in the LSH. Results show that the cosine distance performs better than the Euclidean distance.

B. Codebook sizes. We created codebooks of different sizes and studies their search accuracy. We evaluated the accuracy in terms of the recall rate at two representative false positive rates. Fig. 6b) illustrates the results for the codebooks of 16, 32, 64, and 128 centroids. We can see that the codebook size seems not having a significant influence on the accuracy of Genius. This result provides an insight that allow us to reduce codebook preparation time by using a smaller codebook $n = 16$.

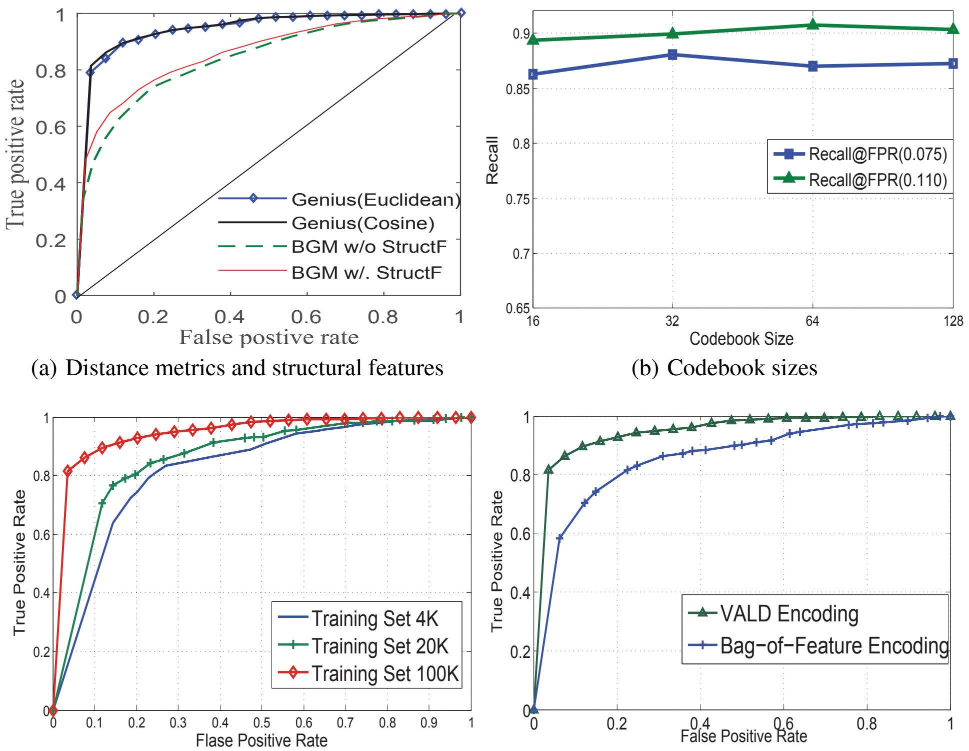
C. Training data sizes. Another important parameter is the size of training set used to generate codebook. We selected training data samples of different sizes to generate the codebook for search. Fig. 6c) shows their search results. We can see that the more samples used for training, the better Genius performed, but the increase in accuracy becomes saturated when the training data is sufficiently large, in our case up to 100 thousand functions. This is consistent with observations from image retrieval methods.

D. Feature encoding methods. We discussed two feature encoding methods in Section 4.2: bag-of-feature and VLAD encoding. We compared their impacts on the search accuracy while fixing other parameters. Fig. 6d) illustrates the ROC curves using two encoding methods. As we can see, VALD performs better than Bag-of-Feature encoding. This observation suggests considering the first-order statistics is beneficial for bug search problem. As the computational cost is similar between VALD and bag-of-features, we recommend using VALD feature encoding in practice.

5.5 Bug Search at Scale

We evaluated the scalability of Genius on Dataset III, which consists of 8,126 firmware images containing 420,336,846 functions, in terms of the preparation phase and search phase. We investigated the time consumption for each stage to demonstrate that Genius is capable of handling firmware images at a large scale.

We encoded 1 million functions randomly selected from Dataset III and collected the preparation time for each of them. The preparation time included the control flow graph extraction and graph



(a) Distance metrics and structural features

(b) Codebook sizes

(c) Training set size

(d) Feature encoding

Figure 6: **Accuracy comparison with different parameter settings. a), c) and d) are ROC curves**

encoding time. Fig. 10a) demonstrates the Cumulative Distribution Function (CDF) of time consumption for randomly selected 1 million query functions. We can see that nearly 90% of the functions were encoded in less than 0.1 seconds. Additionally, less than 10% of the functions needed more than 4 seconds to encode. This is because these functions have more than 1000 basic blocks, and thus take longer to encode. The prepartion time across different sizes of ACGFs is illustrated in Fig. 10b).

We further evaluated the search time for Genius in the large scale codebase. We partitioned Dataset III into six codebases of different scales from $s = 10^3$ to $s = 10^8$, where s is the total number of functions in the codebase. Genius was tested against 1 to 10,000 sequentially submitted queries. Fig. 10c) shows the log-log plot of the time consumption for Genius at the online search phase. As we can see, the search time grows sublinearly according to the increase of the codebase size, and the average search time over observed was less than 1 second for a firmware codebase of about 100 million functions.

5.6 Case Studies

We also evaluated the efficacy of Genius in real bug search scenarios. Case studies were conducted on Genius for the two use scenarios discussed in Section 2. With the aid of case studies, we demonstrated how Genius would work in the real world to facilitate the vulnerability identification process.

Scenario I. In this scenario, we conducted a vulnerability search on Dataset III of 8,126 images using vulnerability queries extracted from Dataset IV. We performed a comprehensive search for two vulnerabilities (*CVE-2015-1791* and *CVE-2014-3508*), which took less than 3 seconds. We then manually verified the vulnerability authenticity for the returned candidate functions. We disassembled the binary code for each candidate, and looked into their semantics to check whether they were patched or not. Due to the workload of manual analysis, we only verified the top 50 candidates for the two selected vulnerabilities. We found 38 potentially vulnerable firmware devices across 5 vendors, and confirmed that 23 were actually vulnerable. We also contacted these product vendors for further confirmation. The following gives the detailed discussion about search results.

CVE-2015-1791. This vulnerability allows remote attackers to cause a denial of service (double free and application crash) on the device. In the top 50 candidates, we found that there were 14 firmware images potentially affected by this vulnerability. We were able to confirm that 10 of these images were actually vulnerable. These images were from two vendors: D-LINK and Belkin.

CVE-2014-3508. This vulnerability allows context-dependent attackers to obtain sensitive information from process stack memory by reading the output of sensitive functions. We found that there were 24 firmware images which could have this vulnerability, and we were able to confirm that the vulnerabilities existed in 13 images from three vendors. These vendors included CenturyLink, D-Link and Actiontec.

This clearly demonstrated that a security evaluator, after only 3 seconds, could get a list of candidate functions to prioritize their search for vulnerable device firmware.

Scenario II. We chose the two latest commercial firmware images from D-Link DIR-810 model as our evaluation targets. We built the LSH indexes for these two firmware images and then searched those two images for all 185 vulnerabilities from Dataset IV (discussed in Section 5.2). It took less than 0.1 second on average to finish searching for all 154 vulnerabilities. We conducted manual verification at the top 100 candidates for each vulnerability and found 103 potential vulnerabilities in total for two images, 16 of

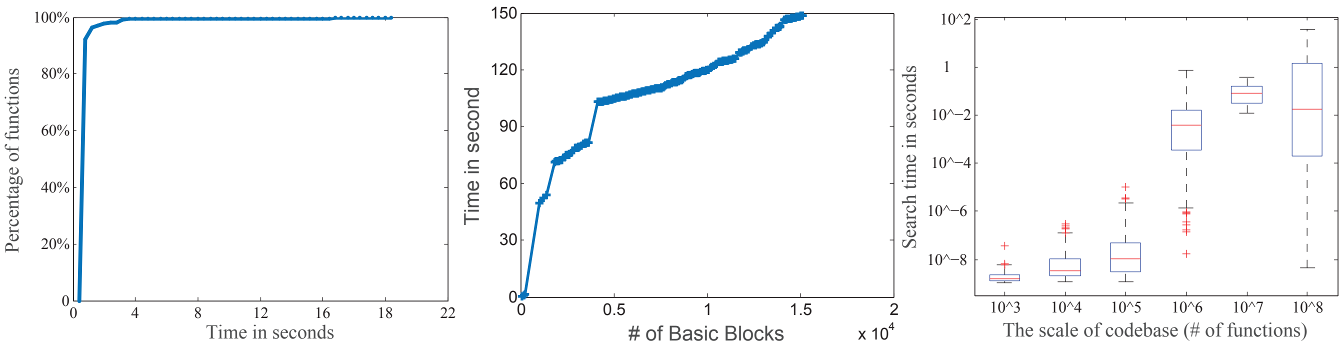


Figure 7: The CDF of preparation time over#1 million functions

Figure 8: The preparation time cross different size of CFG

Figure 9: The search time crossscales of fimware codebases(# of funcitons)

Figure 10: **The breakdown of the performance for** Genius**.**

Table 4: Case study results for Scenario II

DIR-810L_REVB_FIRMWARE_2.03B02			DIR-810L_REVB_FIRMWARE_2.02.B01		
CVE	**Patched**	**Vulnerability Type**	**CVE**	**Patched**	**Vulnerability Type**
CVE-2016-0703	No	Allows man-in-the-middle attack	CVE-2015-0206	No	Memory consumption
CVE-2015-1790	No	NULL pointer dereference	CVE-2014-0160	Yes	Heartbleed
CVE-2015-1791	Yes	Double free	CVE-2015-0289	No	NULL pointer dereference
CVE-2015-0289	No	NULL pointer dereference	CVE-2016-0797	No	Heap memory corruption
CVE-2014-8275	No	Missing sanitation check	CVE-2016-0798	No	Memory consumption
CVE-2015-0209	No	Use-after-free	CVE-2014-3513	No	Memory consumption
CVE-2015-3195	No	Mishandles errors	CVE-2014-3508	No	Information leakage
#	#	#	CVE-2015-0206	No	Memory consumption
#	#	#	CVE-2014-8275	No	Missing sanitation check

which were confirmed (see Table 4). We contacted the product vendor for further confirmation.

Overall, these two case studies substantiate that Genius is an effective tool to facilitate IoT firmware bug searching process for security evaluators.

6. DISCUSSION

While we have demonstrated the efficacy of Genius for accurate, scalable bug search in IoT devices, there are several relevant technical limitations. Our method utilizes static analysis to extract syntactical features, and thus cannot handle obfuscated code which is used to avoid similarity detection (e.g., malware).

Additionally, the accuracy of Genius heavily relies on the quality of CFG extraction. Although IDA pro [28] provides us reasonable accuracy in our evaluation, we can rely on more advanced techniques to further improve its accuracy such as [51].

Furthermore, the accuracy of Genius could be impacted by function inlining, since it may change the CFG structures. Since our main focus in this paper is to improve the scalability of existing in-depth bug search, we will leave the evaluation of Genius for this case as future work.

Like other CFG-based code search approaches, the accuracy of Genius is also affected by the size of the CFG. The smaller the size of CFG is, the more likely it is to have collisions. To be aligned with other work [23], we also considered functions with at least five basic blocks. We believe that this is a reasonable assumption since small functions have significantly lower chance to contain vulnerabilities in a real-world scenario [37].

7. RELATED WORK

We have discussed closely related work throughout the paper. In this section, we briefly survey additional related work. We focus on approaches using code similarity to search for known bugs. There are many other approaches that aim at finding unknown bugs, such as fuzzing or symbolic execution [11, 15, 17, 48, 52, 55] etc. Since they are orthogonal to our approach, we will not discuss these approaches in this section.

Source-Level Bug Search. Many works focused on finding code clones at the source code level. For example, [58] generates a code property graph from the source code and conducts a graph query to search for code clones with the same pattern. Similarly, token-based approaches such as CCFinder [33] and CP-Miner [35] utilize token sequence and scan for duplicate token sequences in other source code. DECKARD [18] generates numerical vectors based upon abstract syntax trees and conducts code similarity matching for code clone detection. ReDeBug [29] provides an efficient and scalable search to find unpatched code clones in OS-distribution bases. All of these approaches require source code, and cannot find bugs in firmware images unless the source code is available.

Binary-Level Bug Search. Since we do not always have access to firmware source code, bug search techniques that work on binary code are very important. One common issue with the current approaches is that they only support a single architecture. It is common that bugs from firmware images in x86 can appear in images of another architecture such as MIPS or ARM, so finding bugs in firmware images demands the capability to handle binary code in a cross-architecture setting.

For example, the tracelet-based approach [20] captures execution sequences as features for code similarity checking, which can defeat the CFG changes. However, the opcode and register names are different across architectures, so it is not suitable for finding bugs in firmware images cross architectures. Myles et al. [41] uses k-grams on opcodes as a software birthmark technique. TEDEM [46] captures semantics using the expression tree of a basic block. The opcode difference on different architectures will easily defeat these two approaches. Rendezvous [34] first explored the code search in binary code. However, it has two limitations. It relies on n-gram features to improve the search accuracy. Secondly, it decomposes the whole CFG of a function into subgraphs. Our evaluation demonstrates that two CFGs as a whole by graph matching is much more accurate than comparing their subgraphs since one edge addition will introduce great difference on the number of subgraphs for two equal CFGs. Therefore, subgraph decomposition will reduce the search accuracy. Finally, as with the other approaches described thus far, it is designed for a single architecture.

Control flow graph (CFG)-based bug searching is a prevalent approach for finding bugs in firmware images. However, most existing works focus on how to improve the matching accuracy by selecting different features or matching algorithms. Flake et al. [25] proposed to match CFGs of a function to defeat some compiler optimizations such as instruction reordering and changes in register allocation. However, the approach relies upon exact graph matching which is too expensive to be applied for large scale bug search. Pewny et al. [45] use I/O pairs to capture semantics at the basic-block level for code similarity computation. It is still expensive for feature extraction and graph matching. DiscovRe [23] utilizes the pre-filtering to facilitate CFG based matching, but our evaluation demonstrates that the pre-filtering is unreliable and outputs tremendous false negatives. Zynamics BinDiff [21] and BinSlayer [13] use a similarity metric based on the isomorphism between control flow graphs to check similarity of two binaries. They are not designed for bug search, especially for finding bug doublets across different binaries where the CFGs of two binaries are totally different. Besides, BinHunt [26] and iBinHunt [39] utilize symbolic execution and a theorem prover to check semantic equivalence between basic blocks. These two approaches are expensive and cannot be applied for large scale firmware bug search since they need to conduct binary analysis to extract the equations and conduct the equivalence checking.

The field of automatic large-scale firmware analysis has also made a breakthrough. Costin et al. [19] carried out an analysis of over 30,000 firmware samples, but it does not perform in-depth analysis. Instead, it extracts each firmware sample and investigates it for artifacts such as private encryption keys. Therefore, this approach is not suitable for finding more general vulnerabilities without these obvious artifacts.

Dynamic analysis based bug search in firmware images. Blanket-execution [22] uses the dynamic run-time environment of the program as features to conduct the code search. This approach can defeat the CFG changes, but it is only evaluated in a single architecture. Besides, dynamic analysis to support firmware images is at the initial stage [17, 61], and still has not been demonstrated its effectiveness with respect to the run-time environments of programs for large scale firmware images.

8. CONCLUSIONS

In this paper, inspired by the image retrieval approaches, we proposed a numeric-feature based search technique to address the scalability issues in existing in-depth IoT bug search approaches. We proposed methods to learn higher-level features from the raw features (control flow graphs), and performed search based upon the learned feature vector rather than directly performing pair-wise matching. We have implemented a bug search system (`Genius`), and compared `Genius` with the state-of-the-art bug search approaches. The extensive experimental results show that `Genius` can achieve even better accuracy than the state-of-the-art methods, and is orders of magnitude faster than most of the existing methods. To further demonstrate the scalability, `Genius` was evaluated on 8,126 devices of 420 million functions across three architectures and 26 vendors. The experiments show that `Genius` can finish a query less than 1 second on average.

Acknowledgment

We would like to thank anonymous reviewers for their feedback. This research was supported in part by National Science Foundation Grant #1054605, Air Force Research Lab Grant #FA8750-15-2-0106, and DARPA CGC Grant #FA8750-14-C-0118. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] Cybersecurity and the Internet of Things. http://www.ey.com/Publication/vwLUAssets/ EY-cybersecurity-and-the-internet-of-things.pdf.

[2] DDWRT ftp. http://download1.dd-wrt.com/dd-wrtv2/ downloads/others/eko/BrainSlayer-V24-preSP2/.

[3] Industrial Utilities and Devices Where the Cyber Threat Lurks. http://www.cyactive.com/ industrial-utilities-devices-cyber-threat-lurks/.

[4] Iot when cyberattacks have physical effects. http: //www.federaltimes.com/story/government/solutions-ideas/ 2016/04/08/internet-things-when-cyberattacks-have\ discretionary{-}{}{}physical-effects/82787430/.

[5] mongodb. https://www.mongodb.com.

[6] Nearpy. https://pypi.python.org/pypi/NearPy.

[7] DD-WRT Firmware Image r21676. ftp://ftp.dd-wrt.com/others/eko/BrainSlayer-V24-preSP2/ 2013/05-27-2013-r21676/senao-eoc5610/linux.bin, 2013.

[8] ReadyNAS Firmware Image v6.1.6. http://www.downloads.netgear.com/files/GDC/ READYNAS-100/ReadyNASOS-6.1.6-arm.zip, 2013.

[9] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM Commun.*, 51, 2008.

[10] R. Arandjelovic and A. Zisserman. All about vlad. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1578–1585, 2013.

[11] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[12] M.-F. Balcan, A. Blum, and A. Gupta. Approximate clustering without the approximation. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1068–1077, 2009.

[13] M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[14] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern recognition letters*, 19(3):255–259, 1998.

[15] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Oakland*, 2015.

[16] K. Chatfield, V. S. Lempitsky, A. Vedaldi, and A. Zisserman. The devil is in the details: an evaluation of recent feature encoding methods. In *BMVC*, volume 2, page 8, 2011.

[17] D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[18] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, 2015.

[19] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security*, 2014.

[20] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[21] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[22] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security*, 2014.

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

[24] Q. Feng, A. Prakash, M. Wang, C. Carmony, and H. Yin. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *ASIACCS*, 2016.

[25] H. Flake. Structural comparison of executable objects. In *DIMVA*, volume 46, 2004.

[26] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*. 2008.

[27] J. Holcombe. Soho network equipment (technical report). https://securityevaluators.com/knowledge/case_studies/routers/soho_techreport.pdf.

[28] The IDA Pro Disassembler and Debugger. http://www.datarescue.com/idabase/.

[29] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Oakland*, 2012.

[30] L. Jiang, T. Mitamura, S.-I. Yu, and A. G. Hauptmann. Zero-example event search using multimodal pseudo relevance feedback. In *ICMR*, 2014.

[31] L. Jiang, W. Tong, and A. G. Meng, Deyu andHauptmann. Towards efficient learning of optimal spatial bag-of-words representations. In *ICMR*, 2014.

[32] L. Jiang, S.-I. Yu, D. Meng, T. Mitamura, and A. G. Hauptmann. Bridging the ultimate semantic gap: A semantic search engine for internet videos. In *ICMR*, 2015.

[33] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[34] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013.

[35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, volume 4, pages 289–302, 2004.

[36] W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In *ICML*, 2011.

[37] McCabe. More Complex = Less Secure. Miss a Test Path and You Could Get Hacked. http://www.mccabe.com/sqe/books.htm, 2012.

[38] A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *the workshop on learning for text categorization*, 1998.

[39] J. Ming, M. Pan, and D. Gao. ibinhunt: binary hunting with inter-procedural control flow. In *Information Security and Cryptology*, pages 92–109. Springer, 2012.

[40] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[41] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

[42] M. Newman. *Networks: an introduction*. 2010.

[43] A. Y. Ng, M. I. Jordan, Y. Weiss, et al. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 2:849–856, 2002.

[44] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *CCS*, 2015.

[45] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Oakland*, 2015.

[46] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging semantic signatures for bug search in binary programs. In *ACSAC*, 2014.

[47] G. Qian, S. Sural, Y. Gu, and S. Pramanik. Similarity between euclidean and cosine angle distance for nearest neighbor queries. In *Proceedings of the symposium on Applied computing*, pages 1232–1237, 2004.

[48] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *USENIX Security*, 2014.

[49] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and vision computing*, 27(7):950–959, 2009.

[50] M. Shahrokh Esfahani. Effect of separate sampling on classification accuracy. *Bioinformatics*, 30:242–250, 2014.

[51] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security*, 2015.

[52] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[53] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *IEEE International Conference on Computer Vision*, 2003.

[54] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *Signal Processing Magazine, IEEE*, 25(2):128–131, 2008.

[55] N. Stephens, J. Grosen, C. Salls, A. Dutcher, and R. Wang. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[56] M. Wall. Galib: A c++ library of genetic algorithm components. *Mechanical Engineering Department, Massachusetts Institute of Technology*, 87:54, 1996.

[57] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

[58] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Oakland*, 2015.

[59] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo. Evaluating bag-of-visual-words representations in scene classification. In *International workshop on Workshop on multimedia information retrieval*, 2007.

[60] S.-I. Yu, L. Jiang, Z. Xu, Y. Yang, and A. G. Hauptmann. Content-based video search over 1 million videos with 1 core in 1 second. In *ICMR*, 2015.

[61] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.

[62] M. Zhang, Y. Duan, Q. Feng, and H. Yin. Towards automatic generation of security-centric descriptions for android apps. In *CCS*, 2015.

[63] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *CCS*, 2014.