BinArm: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices

Paria Shirani[1], Leo Collard[1], Basile L. Agba[2], Bernard Lebel[3]
Mourad Debbabi[1], Lingyu Wang[1], and Aiman Hanna[1]

[1] Security Research Centre, Concordia University, Montreal, Canada
[2] Institut de recherche d'Hydro-Québec, Montreal, Canada
[3] Thales Canada Inc., Montreal, Canada

Abstract. There is a widespread adoption of intelligent electronic devices (IEDs) in modern-day smart grid deployments. Consequently, any vulnerabilities in IED firmware might greatly affect the security and functionality of the smart grid. Although general purpose techniques exist for vulnerability detection in firmware, they usually cannot meet the specific needs, e.g., they lack the domain knowledge specific to IED vulnerabilities, and they are often not efficient enough for handling the larger firmware of IEDs. In this paper, we present BinArm, a scalable approach to detecting vulnerable functions in smart grid IED firmware mainly based on the ARM architecture. To this end, we build comprehensive databases of vulnerabilities and firmware that are both specific to smart grid IEDs. Then, we propose a multi-stage detection engine to minimize the computational cost of function matching and to address the scalability issue in handling large IED firmware. Specifically, the proposed engine takes a coarse-to-fine grained multi-stage function matching approach by (i) first filtering out dissimilar functions based on a group of heterogeneous features; (ii) further filtering out dissimilar functions based on their execution paths; and (iii) finally identifying candidate functions based on fuzzy graph matching. Our experiments show that BinArm accurately identifies vulnerable functions with an average precision of 0.92 and recall of 0.83. The experimental results also show that our detection engine can speed up the existing fuzzy matching approach by three orders of magnitude. Finally, as a practical tool, BinArm successfully detects 93 real-world CVE vulnerability entries, the majority of which have been confirmed, and the detection takes as little as 0.09 seconds per function on average.

1 Introduction

Intelligent electronic devices (IEDs) play an important role in typical smart grids by supporting SCADA communications, condition-based monitoring, and polling for event-specific data in the substations. The firmware (software) running on IEDs is subject to a wide range of software vulnerabilities, and consequently security attacks exploiting such vulnerabilities may have debilitating repercussions on national economic security and national safety [1]. In fact, a startling increase in the number of attacks against Industrial Control System (ICS) equipment has been observed (e.g., a 110% increase when comparing 2016 to 2015 [15]). A prime example of such an attack is *Industroyer* [3] targetting Ukraine's power grid, which is capable of directly controlling electricity substation switches and circuit breakers. As other examples, the *Black Energy* [46] APT took control of operators' control stations, and utilized them to cause a blackout; and *Stuxnet* [29, 42] targeted Siemens ICS equipment in order to infiltrate Iranian nuclear facilities. In addition to those real-world attacks, industrial analysis demonstrate similar threats in other countries, e.g., with 50 power generators taken over by attackers, as many as 93 million US residents may be left without power [53]. These real-world attacks or hypothetical scenarios indicate a clear potential and serious consequences for future attacks against electrical infrastructures including smart grids.

Identifying security-critical vulnerabilities in firmware images running on IEDs is essential to assess the security of a smart grid. However, this task is especially challenging since the source code of firmware is usually not available. In the literature, general purpose techniques have been developed to automatically identify vulnerabilities in embedded firmware based on dynamic analysis (e.g., [19, 24, 55, 61]) or static analysis (e.g., [22, 28, 30, 50, 60]). To the best of our knowledge, none of the existing approaches focuses on the smart grid context. Although such general purpose techniques are also applicable to the firmware of smart grid IEDs, they share some common limitations as follows. (i) *Applicability*: They lack sufficient domain knowledge specific to smart grids and IEDs, such as a database of known vulnerabilities in such devices and that of the IED firmware. The general purpose approaches (a)can easily crawl and download any firmware from the wild, without requiring any prior knowledge about the scope, (b) do not put additional effort to gather and analyse the relevant IED firmware images, (c) do not study the used libraries in the IED firmware images; high likely most relevant libraries are not included in their vulnerability dataset, which might result in more false negative rates. (ii) *Scalability*: Those approaches typically rely on expensive operations, such as semantic hashing [50], and they typically lack effective filtering steps to speed up the function matching. Consequently, those techniques are usually not efficient enough to handle the much larger sizes of IED firmware (e.g., compared to that of network routers) and not scalable enough for a large scale application to real-world smart grids. (iii) *Adaptability*: Handling the presence of a new CVE and efficiently indexing it poses another challenge to some existing works (e.g., [60]).

In this paper, we present BinArm, a scalable approach to detecting vulnerable functions in smart grid IED firmware mainly based on the ARM architecture. To this end, we first build a large-scale vulnerability database consisting of common vulnerabilities in IED firmware images. The design of our vulnerability database is highly influenced and guided by the prominent libraries used in the IED firmware images. To identify these IEDs and obtaining the corresponding firmware images we dedicate significant efforts to: (i) identify relevant manufacturers, (ii) collect and analyze the corresponding IED firmware images,

(iii) identify the used libraries in these images, (iv) compile the list of CVE vulnerabilities, and push them to the vulnerability database. Such effort can be leveraged for future research on smart grid IEDs, and can be beneficial to IED vendors as well as utilities to assess the security of elaborate/deployed IED firmware.

Second, to ensure BinArm is efficient and scalable enough to handle IED firmware images, we design a detection engine that employs three increasingly complex stages in order to speed up the process by filtering out mismatched candidates as early as possible. Third, BinArm does not only provide a similarity score as prior efforts, such as [28, 60], rather presents in-depth (at instruction, basic block and function levels) details to justify the results of the matching and to assist reverse engineers for further investigation. We conduct extensive experiments with a large number of real-world smart grid IED firmware from various vendors in order to evaluate the effectiveness and performance of BinArm.

Contributions. Our main contributions are as follows:

- To the best of our knowledge, we develop the first large scale vulnerability database specifically for IEDs firmware covering most of the major vendors (e.g., SEL). In addition, we build the first IED firmware database, which gives an overview of the state of the industry.
- We propose a multi-stage detection engine to efficiently identify vulnerable functions in IED firmware, while maintaining the accuracy. The experiments demonstrate this engine is three orders of magnitude faster than the existing fuzzy matching approach [36].
- Our experimental results ascertain the accuracy of the proposed system, with an average precision of 0.92 and recall of 0.83. In addition, the real-world applicability of BinArm is confirmed in our study, which successfully detects 93 potential CVEs among real-world IED firmware within 0.09 seconds per function on average, the majority of which have been confirmed by our manual analysis.

2 Approach Overview

An overview of our approach is depicted in Figure 1, which consists of two major phases: *offline preparation* and *online search*. The *offline preparation* phase consists in the creation of two comprehensive databases; one containing a set of IED firmware and the other known vulnerabilities specific to IEDs. To this end, we:

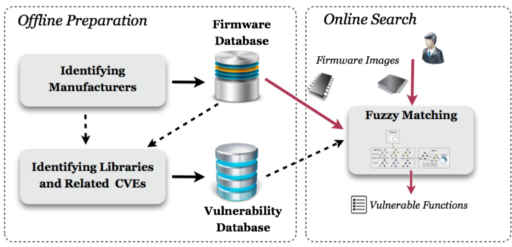


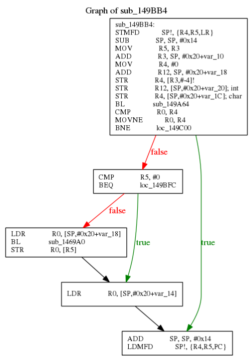
Fig. 1: BinArm overview

- Identify a set of manufacturers that provides equipment for smart grids.
- Collect relevant IED firmware produced by the identified manufacturers, and store the images in the *Firmware Database*.

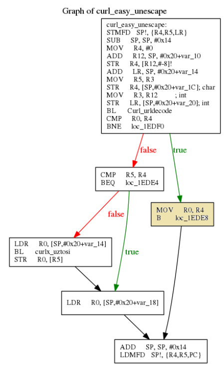
Such information further provides insight about which libraries might be utilized by each manufacturer in their released firmware, which enables us to build our vulnerability database. For this purpose, we

- Determine reused libraries in the IED firmware from manufacturers' websites or available documentations, e.g, the copyright provided by NI [11].
- Collect the identified open-source and vulnerable libraries, and cross-compile them for the ARM processor in order to build the *Vulnerability Database*.

We demonstrate how the aforementioned process works by applying it to a motivating example in the following. Suppose a fictitious utility company has recently deployed several phasor measurement units (PMUs) and is concerned about potential vulnerabilities inside those units. Following our methodology depicted in Figure 1, we would first identify the manufacturer, e.g., given by the utility as National Instruments (NI) in this particular example. Second, we would collect the IED firmware, which is again given by the utility as the NI PMU1_0_11 firmware image [12]. Third, we would identify the reused libraries in this firmware, e.g., the `libcurl` v7.50.2 library. Fourth, we would identify vulnerable functions inside each library, e.g., a vulnerable function inside the `libcurl` v7.50.2 library as depicted in Figure 2b. Finally, we employ our detection engine to find matching functions in the provided firmware image, e.g., a matching function is shown in Figure 2a. As shown, the two functions have a high degree of similarity; indeed, the only difference is the presence of an additional basic block consisting of two instructions (highlighted in Figure 2b) in the `curl_easy_unescape` function. This similarity implies that the function in Figure 2a may have the CVE-2016-7167 vulnerability, which provides useful information for the utility company to take corresponding actions.



(a) **`sub_149BB4`** CFG in `NI PMU1_0_11.libws_repl`



(b) **`curl_easy_unescape`** CFG in `libcurl` v7.50.2 with CVE-2016-7167

Fig. 2: An example of function reuse in IED firmware

We note that, although this particular example may make it seem relatively straightforward to detect vulnerable functions in a firmware, this is usually not the case in practice due to two main challenges. First, the needed information about manufacturer, libraries, and vulnerabilities may not be readily available from the utility company as in this example. For this reason, we will build our vulnerability and firmware databases in Section 3. Second, the function matching process may be too expensive for utility companies, since they may be dealing with the constant deployment or upgrade of thousands of IEDs from different manufacturers, and cross checking such a large number of firmware images[4] with

[4] *Linksys WRT32X* with $39kb$ size contains $47,025$ functions, whereas *NI PMU1_0_11* firmware comprises $226,496$ functions and is $256kb$ large.

an even larger number of library functions (e.g., $5,103$ vulnerable functions) can take significant effort. To address this challenge, we will propose our efficient multi-stage detection engine in Section 4.

3 Building IED Firmware and Vulnerability Databases

Identifying the IEDs and obtaining the corresponding firmware images significantly require more effort, compared to acquiring firmware of many consumer devices that can be easily crawled and downloaded from the wild. To the best of our knowledge, our database of smart grid and IED-specific firmware vulnerabilities is the first such effort. It can be leveraged for future research on smart grid IEDs. In addition, it could help vendors and utilities in assessing the security of the elaborated or deployed IED firmware. In this section, we first introduce the smart grid IEDs, and then discuss how the contents of our firmware and vulnerability databases have been established.

3.1 Intelligent Electronic Devices (IEDs) in Smart Grid

A power grid is a complex and critical system to provide generated power to a diverse set of end users, which is composed of three main sectors: generation, transmission and distribution. The role of a distribution substation is to transform received high voltage electricity to a lower more suitable voltage for distribution to customers. The IEC 61850 [2] standard is introduced to leverage technologies, such as Ethernet, high speed wide area networks (WAN), and powerful but cheap computers to define a modern architecture for communication within a substation [45]. Consequently, a vast set of devices labelled as intelligent electronic devices (IEDs) are emerged, which are coupled with traditional ICS and power equipment to enable their integration into the network.

An IED can belong to three general non-exclusive categories: (i) *Control*: send and receive commands to control the system behaviour remotely, such as bay, load-shedding, circuit breaker, and switch; (ii) *Monitoring and relay*: convert received analog input (e.g., currents, voltages, power values) from primary equipment into a digital format that can be used throughout the network, such as phasor measurement units (PMUs), and phasor data concentrators (PDCs); and (iii) *Protection*: detect faults that need to be isolated from the network in a specific and timely manner, such as busbar, generator, line distance and breaker.

3.2 Manufacturer Identification

To identify a set of recognized manufacturers that are relevant to the smart grid and its components, we first study the categorization of smart grid vendor ecosystem, most relevant vendors, and market dynamics using different sources, such as *GTM Research* [33, 38] and *Cleantech Group* [47] reports. These information provide the necessary insight to identify top smart grid manufacturers, as listed in Table 1. Such knowledge about manufacturers becomes the foundation to further determine relevant libraries, vulnerabilities and IED firmware images.

Table 1: Identified major smart grid manufacturers and their supported components

Relevant Component	**Manufacturer**													
	ABB	*Aclara*	*Cisco*	*EI*	*Elster*	*GE*	*Honeywell*	*Itron*	*L+G*	*NI*	*SE*	*SEL*	*Sensus*	*Siemens*
Automation Hardware	•									•	•	•		•
Smart Meters		•		•	•	•		•	•				•	
Automation Software			•		•				•					
Communication			•		•			•	•				•	
Demand Response							•							

(ABB): ABB Schweiz AG. (EI): Electro Industries. (GE): General Electric. (L+G): Landis+Gear. (NI): National Instruments. (SE): Schneider Electric. (SEL): Schweitzer Engineering Laboratories.

Heterogeneous hardware architectures are used in firmware images, however, many industrial control systems are based on the ARM architecture [18, 41, 64]. Additionally, as reported in Figure 3, most of the collected IED firmware images are identified as targeting ARM architecture (82%), followed by PowerPC (9%). On the other hand, Linux is the most encountered operating system in our firmware dataset, with 90% of frequency amongst other operating systems, such as Windows. Therefore, in this work we focus on the ARM-based and Linux-based IED firmware images.

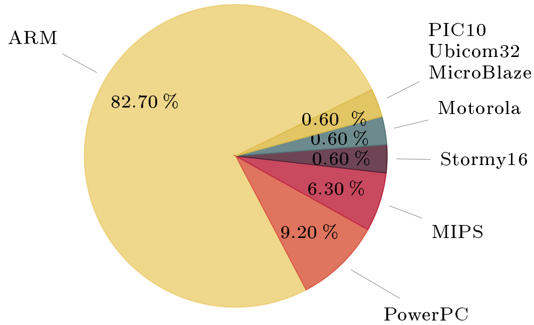


Fig. 3: Distribution of hardware architectures among collected IED firmware

3.3 Vulnerability Database

Our study shows that many of the listed companies reuse existing free open-source software in their product implementations. This generally entails the legal obligation of publishing documents containing the licenses of all utilized open-source software. By investigating several sources of information pertaining to these manufacturers, such as corporate websites, product documentations, and FTP search tools, we extract large amounts of open-source usage declarations that are related to the current smart grid scope, such as simple network management protocol (SNMP), and network time protocol (NTP).

Table 2 illustrates the top 25 relevant, vulnerable, and popular open-source libraries, which are ordered by their relative significance considering which vulnerable libraries are more frequently used in the recognized manufacturers. We download the source code of reused libraries with different versions, and cross-compile them for the ARM architecture using the GCC compiler with four optimization flags ($O0 - O3$). Then we utilize a CVE database [8] to identify the number of publicly known CVEs for each of these libraries. It is worth mentioning that all the functions of each library are stored in our *Vulnerability Database*, and the vulnerable functions are labelled by their corresponding identified CVEs.

Our *Vulnerability Database* consists of 3,270,165 functions, 5,103 of which are marked as vulnerable. This results in a total of 235 unique vulnerabilities after discarding the duplicates that are created due to the use of different compilers and optimization flags.

It has not escaped our notice that the firmware images are composed of various kinds of binaries, such as kernel, application-level, open-source as well as proprietary libraries. Consequently, based on the CVE database [7], we have identified 4344 CVE vulnerabilities in kernel-level, 5581 in application-level, and 2336 in open-source libraries amongst the identified manufacturers, considering the fact that some of the open-source libraries are reused in applications. Additionally, we have prepared an initial list of IED-specific proprietary libraries (e.g., NI). However, our list of such proprietary libraries is not comprehensive yet. Also, additional effort for the verification of vulnerability identification would be required, since the source code of such proprietary libraries are not publicly available. This task remains as the subject of our future work.

3.4 Firmware Database

The proposed methodology is not necessarily specific to smart grid IEDs and therefore could be applied to any ARM-based firmware, such as IoT devices, routers, and IEDs. However, since the goal of this work is to assess the security of IEDs in the smart grid, we focus on firmware images that are directly relevant to IEDs. We first utilize popular FTP search engines to leverage accessible public corporate FTP servers. We then create a simple website scraper [14] and apply it to specific parts of each manufacturers' website. Finally, we perform a manual inspection for dynamically generated websites, which mostly applies to each manufacturers' download centre. All retrieved images are then filtered based on the relevance to smart grid context. In the end, we extract 2,628 firmware packages from the mentioned vendors.

Firmware Analysis Challenges Performing firmware analyses with the objective of complete disassembly is a challenging task [22]. This is partially due to a large requirement of time, domain specific knowledge and research [40]. Furthermore, binaries are often stored in proprietary formats, obfuscated or encrypted for protection. These processes effectively make it extremely difficult (e.g. obfuscation [40]), or even impossible (e.g. uncrackable encryption [57], indecipherable formats [43]) to directly access the contents of a given blob. Encrypted binaries can sometimes be identified by their use of specific headers. For instance, the file encrypted with `openssl` start with the first 8-byte signature of `"Salted__"`. Additionally, a given binary blob can contain several entry points [55], and it may not be possible for tools such as IDA PRO to automatically identify them. In these cases entry point discovery should be performed [55, 40]. This is one of the most challenging parts of this entire procedure and required leveraging various techniques (e.g., [64]). In order to process all acquired firmware, we follow well-known procedures that are presented in [62, 57]. This process has several main steps: (i) unpacking and extraction, which consists of removing all files from another compressed file; (ii) firmware identification, that can be located amongst or within the extracted files; (iii) hardware architecture identification and scanning for op-code signatures to be identified as ARM; (iv) image base identification in order to know where the binary should be loaded ; and (v) disassembling using IDA Pro disassembler [10], where using the properly identified architecture and entry point is required.

Table 2: Vulnerable open-source libraries

Library	**No. CVEs**	**Manufacturers**
php	601	Cisco, Honeywell, Siemens
imagemagick	402	Cisco, GE, Honeywell
openssl	189	ABB, Cisco, GE, Honeywell, Schneider Electric, Siemens
mysql	564	Cisco
tcpdump	162	Cisco, GE, Siemens
openssh	87	ABB, Cisco, GE, Honeywell, Siemens
ntp	79	Cisco, GE, Honeywell, Schneider Electric, Siemens
libtiff	149	Cisco, GE
postgresql	98	Cisco, Honeywell, Siemens
ffmpeg	274	Siemens
pcre	49	ABB, Cisco, GE, Honeywell, Siemens
python	81	Cisco, Honeywell, Siemens
glibc	81	Cisco, Honeywell, Siemens
qemu	225	Cisco
libxml2	44	ABB, Cisco, GE, Honeywell, Siemens
bind	102	Cisco, Siemens
binutils	97	Cisco, Siemens
libcurl	34	ABB, Cisco, Honeywell, Schneider Electric, Siemens
freetype	83	Cisco, Siemens
libpng	47	Cisco, Honeywell, Siemens
samba	124	Honeywell
utillinux	15	ABB, Cisco, GE, Honeywell, Schneider Electric, Siemens
cups	88	Cisco
lighttpd	28	ABB, Cisco, Honeywell
netsnmp	21	Cisco, GE, Schneider Electric, Siemens

4 Multi-stage Detection Engine

We propose a multi-stage detection engine to more efficiently identify vulnerable functions in firmware images, which starts with a coarse detection and moves towards more granular detection stages composed of: (i) function shape-based detection; (ii) branch-based detection; and (iii) fuzzy matching-based detection, The key idea of our multi-stage detection engine is to start with light-weight feature extraction and function matching operations, and to perform the most expensive operations (e.g., graph matching) in the end for selected candidates.

To this end, during the first stage, BinArm extracts the simplest and more distinguishable features that quickly eliminates dissimilar candidates with a very less computational overhead. As a result, the potential candidates can be initially shortlisted based on distances of their basic features. During the second stage, BinArm performs more expensive matching operations, however, still not as expensive as graph matching or neighbourhood exploration. To avoid complex matching comparison, BinArm specifically extracts execution paths including the instruction-set and turns them into hash values, and simply employs a binary search. In the final stage, BinArm performs the most expensive operations, which mainly includes careful examination of instruction-level features, the neighbours, and graph matching for a selected number of candidates to minimize the overhead. The details of each stage is explained in the following.

4.1 Function Shape-based Detection

Inspired by [54], the *shape* of a function could be determined by extracting heterogeneous features at different levels. To capture the topology of a function, we employ a set of graph metric. However, some functions may have the same structural shape, while semantically they are different. As a result, we consider instruction and statistical features in order to include semantic information as well. Consequently, the function shape-based detection is performed based on a collection of heterogeneous features extracted from a function, namely, the function *shape* as explained in the following.

Feature Extraction. The first category of features, *instruction-level features*, carries the syntax and semantic information of a function. For instance, the frequencies of strings have been used to classify malware based on their behaviour [52]. The *structural features* category, includes elements of a function shape derived from its graph metrics [32], which mainly represent the structural properties of a function. Finally, *statistical features* are used in order to capture the semantics of a function [51]; for instance, the skewness and kurtosis [4] are extracted as $S_k = (\frac{\sqrt{N(N-1)}}{N-1})(\frac{\sum_{i=1}^{N}(Y_i-\overline{Y})^3/N}{s^3}), K_z = \frac{\sum_{i=1}^{N}(Y_i-\overline{Y})^4/N}{s^4} - 3$, where N is the number of data points, Y_i is the frequency of each instruction, and $\overline{Y}$ and s represents the *mean* and *standard deviation*, respectively. An excerpt of the extracted features is listed in Table 3. For the sake of space, the details of other features are omitted and can be found in [54].

Table 3: An excerpt of function shape features

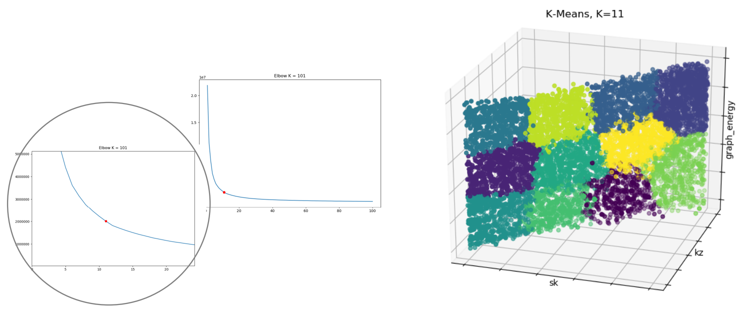
Feature Category	Examples
Instruction-level	#instructions, #arguments, #strings, #mnemonics, #callees, #constants
Structural	#nodes, #edges, cyclomatic complexity, average_path_length, graph_energy, link_density
Statistical	skewness, kurtosis, Z-score, standard deviation, mean, variance

Normalization. In the ARM instruction set, each assembly instruction consists of a mnemonic and a sequence of up to five operands. We may have two fragments of code that are identical both structurally and syntactically, but differ in terms of memory references or registers. Hence, it is essential to normalize the instruction sets prior to comparison. For this purpose, we normalize the operands according to the mapping sets provided by IDA Pro. We further categorize the 'general' registers based on their types.

Feature Selection. In order to identify the most differentiable features, mutual information (MI) [49] is leveraged to measure the dependency degree between the aforementioned features and the functions in the *Vulnerability Database*. Based on the results, we choose three top-ranked features, `graph_energy`, `skewness` (`sk`), and `kurtosis` (`kz`), as a 3-tuple feature for each function. It is worth mentioning that we could consider next two top-ranked features (e.g., rich_club_metric and link_density) as well. However, based on our experiments, there is a dependency between these two features and graph_energy. Additionally, since our goal is to perform coarse detection at this stage, and extracting more features would affect the time complexity, we choose the first three top-ranked features, which the first two are from the statistical category and the latter is part of the structural category. Our experiments confirm the effectiveness of these three feature (Section 5.6).

Function Matching. All functions that surpass a predetermined threshold distance, λ, from a given target function are deemed dissimilar in shape-based detection stage. Euclidean distance of $d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$ is used to calculate the similarity between two functions, where $p = (p_1, p_2, p_3)$ and $q = (q_1, q_2, q_3)$ are 3-tuple associated with each function consisting of graph_energy, sk, and kz features. In order to calculate the threshold distance, we employ K-Means clustering on the extracted features and, based on the distance between the clusters, the final threshold value of $\lambda = 26.45$ is obtained as the following.

Threshold Selection. We acquire the threshold value empirically by leveraging K-Means clustering. K-Means clustering algorithm partitions n observations into k clusters, $C_1, \ldots, C_k$, such that the total within-cluster sum of square $WSS = \sum_{i=1}^{k}\sum_{p \in C_i} dist(p, c_i)^2$ [34] is minimized, where p is representing a given observation; c_i is the centroid of cluster C_i, and $dist$ is the Euclidean distance. To identify the optimal number of clusters, we employ the elbow method [25], where the goal is to get a small WSS while minimizing k. The K-means clustering is applied to our data points for each value of k starting from one to 100, and the WSS is calculated as depicted in Figure 4a. The optimal value for k is at the location of the knee which is equal to 11. The results of eleven clusters are shown in Figure 4b. To achieve the threshold value of λ, first we calculate the average Euclidean distances of all 3-tuple points in each cluster separately to measure how far the similar functions are. Then, the average of eleven obtained distances is calculated and, according to *Vulnerability Database*, the threshold value of $\lambda = 26.45$ is returned.



(a) Elbow results: K=11

(b) K-Means results

Fig. 4: Determining the optimal value of K, and K-Means results

4.2 Branch-based Detection

In the next stage, BinArm incorporates a branch-based detection to reduce the graph comparison effort during the final detection stage. The idea behind branch-based detector is that similar functions have similar execution paths. In addition, analyzing the execution paths has been used to identify function vulnerabilities as well as stealthy program attacks [56, 59].

Weighted Normalized Tree Distance (WNTD). The normalized tree distance (NTD) [63] is proposed for comparing phylogenetic trees with the same topology and same set of N taxonomic groups, as depicted in Figure 5. Consider two trees A and B with the same topology and same set of taxa denoted by $A = \{a_1, a_2, \ldots, a_N\}$ and $B = \{b_1, b_2, \ldots, b_N\}$, where N is equal to path lengths. In order to compare trees A and B, the distance is measured as $NTD = \frac{1}{2}(\sum_{i=1}^{N} |\frac{a_i}{\sum_{j=1}^{N} a_j} - \frac{b_i}{\sum_{j=1}^{N} b_j}|)$ [63], where a_i and b_i are the lengths of path i from trees A and B, respectively. Such a dissimilarity metric scales from 0 (identical trees) to 1 (distinct trees). However, NTD is originally designed for two trees with the same topology (the same number of paths). Additionally, NTD does not consider the contents of nodes.

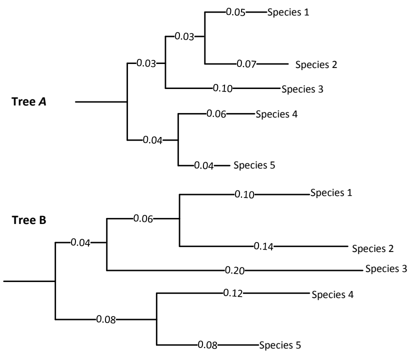


Fig. 5: An example of two trees with the same topology

Therefore, we propose a weighted normalized tree distance (WNTD) metric to measure dissimilarity between two functions W and V. First, we represent the CFGs as a directed acyclic graph, and then all possible paths are extracted from the two CFGs using breadth first search. Based on the contents of basic blocks along the path and their neighbours, a weight is calculated (which will be discussed later) and assigned to each path. The dissimilarity between $W = \{w_1, w_2, \ldots, w_N\}$ and $V = \{v_1, v_2, \ldots, v_M\}$ functions, containing N and M ($N \leq M$) number of weights representative of each path (which is called "weighted paths"), is measured as the following:

$$WNTD = \frac{1}{2}(\sum_{i=1}^{N} |\frac{w_i}{\sum_{j=1}^{N}(w_j)} - \frac{v_{BM}}{\sum_{j=1}^{M}(v_j)}|)$$

where w_i and v_i are the weighted paths in functions W and V, respectively; and v_{BM} is the best match for weighted path w_i amongst the other weighted paths in function V as the following:

$$v_{BM} = \begin{cases} excatMatch(w_i, \boldsymbol{V}) & \text{, if there is any exact match} \\ inexactMatch(w_i, \boldsymbol{V}, \delta) & \text{, if there is any inexact match} \leqslant \delta \\ 0 & \text{, else} \end{cases} \quad (1)$$

WNTD considers a weight for each node (basic block) and finally a single weight for each path of a function. Moreover, even if the two CFGs do not have the same number of paths, it can still find a match for that path as either the best match or zero. Once the WNTD comparison is performed, the functions with a distance less than γ are preserved for the final detection step. Our experiments suggest 50% cut off is the best.

Mnemonic Instructions Grouping. Instruction mnemonics carry information about the semantics of a function, for instance, cryptographic functions perform more logical and mathematical operations compared to a function which opens a file. However, due to different factors, such as compiler effects, various mnemonics might be used interchangeably. Therefore, we identify the list of ARM instruction sets [5], and group them based on their functionality, e.g., arithmetic instructions. As a result, we obtain seventeen groups of mnemonics and then the mutual information (MI) is leveraged to measure the dependency degree between mnemonic group frequencies and functions in our Vulnerability dataset. Accordingly, we choose the 7-top-ranked mnemonic groups as the final features to be extracted from each basic block in a path.

Algorithm 1: Weight Assignment

Input: $Path_a$: A path extracted from the CFG.
Output: w : Branch weight.
Initialization
$f[] \leftarrow 0;$ \\ PDF of top-ranked instruction groups;
$weights[] \leftarrow 0;$ \\ Feature vector of the weights;
$w \leftarrow 0;$ \\ Initialize the path weight to zero;
begin
  foreach $node[i] \in Path_a$ **do**
    $f \leftarrow node[i].getPDF()$; $U[] \leftarrow 0$; $J[] \leftarrow f$;
    while *(node[i].hasParents())* **do**
      $J \leftarrow J \cap node[i][].getParent().getPDF()$;
    end
    while *(node[i].hasChildren())* **do**
      $U \leftarrow U \cup node[i][].getChild().getPDF()$;
    end
    $f \leftarrow J + U$; $weights[i] \leftarrow TLSH(f)$;
  end
  foreach $wt[i] \in weights$ **do**
    $w \leftarrow w + wt[i]$;
  end
  return w;
end

Algorithm 2: WNTD

Input: $W[]$: Branch weights of function W stored in a linked list.
Input: $BTree_V$: Branch weights of function V stored in a B$^+$tree.
Output: $WNTD$: Dissimilary score between functions W and V.
Function WNTD(W,$BTree_V$)
  $sum \leftarrow 0$; $sum_W \leftarrow \sum_{j=1}^{N}(w[j])$;
  $sum_V \leftarrow \sum_{j=1}^{M}(v[j])$;
  foreach $w[i] \in W$ **do**
    v_{BM} = exactMatch($BTree_V$, w_i) ;
    if v_{BM} *!= -1* **then**
      $sum+ = |\frac{w[i]}{sum_W} - \frac{v_{BM}}{sum_V}|$; ;
      $W.remove(w[i])$;
    end
  end
  $v_{BM} \leftarrow 0$;
  foreach $w[i] \in W$ **do**
    v_{BM} = inexactMatch($BTree_V$, w_i, δ); ;
    $sum+ = |\frac{w[i]}{sum_W} - \frac{v_{BM}}{sum_V}|$
  end
  return $WNTD = sum/2$;
end

Weight Assignments. To condense all the information of a node and its neighbours into a single hash value, a graph kernel with linear time complexity is proposed in [31, 35]. Inspired by this approach, we calculate the accumulated weights of each node along the path and assign a single hash value to each path. The weight assigned to each node is calculated based on the top-ranked instruction groups of the node itself and its neighbours (parents and children) that could be out of the current path. For this purpose, we first extract the top-ranked instruction groups and create a feature vector of their probability density function (PDF) for each node and its neighbours. We further distinguish between the in-degrees (parents) and out-degrees (children) by calculating the joint and the union of the PDFs, respectively. Finally, TLSH [48] is applied on the obtained feature vector and a weight is assigned to each node. This process is performed on all the nodes in a given path, and the final weighted path is obtained by the summation of all hash values along the path. The details are presented in Algorithm 1.

Finding the Best Match. In order to find the best match for each path, we pre-calculate all the weights of all paths for both reference and target functions foremost, and then store the obtained weighted paths of the larger function V in a B^+tree. Afterwards, we perform exact and inexact matching to acquire the best match for weighted paths. First, we search in the B^+tree to find the exact match for each weight in function W, and then remove it from the B^+tree. Second, we perform inexact matching by considering *backward* and *forward* sibling pointers to each leaf node, which points to the previous and next leaf nodes, respectively. The number of neighbours is obtained by a user-defined distance δ (Equation 1). If there is not any match for a given path, the best match would be zero. The details of calculating the WNTD is presented in Algorithm 2. The time complexity to find the best match is $O(n \log m)$.

4.3 Fuzzy Matching-based Detection

The results of the branch-based detection stage, which are a relatively small set of candidate functions, are passed to the final detection stage. In order to compare a given target function with the reference functions in the candidate set, inspired by [36], we perform fuzzy matching on each pair of functions and obtain the similarity score. Functions with the highest similarity scores are returned as the final matching pairs. The details are described in the following.

Path and Neighbourhood Exploration. The fuzzy matching approach is composed of three main phases: (i) longest path extraction; (ii) path exploration; and (iii) neighbourhood exploration, which is illustrated with an example in Figure 6. First, we unroll all the loops and employ depth first search on the CFG of target function to extract the longest path (as depicted in Figure 6 part a). A path represents one complete particular execution, where its functionality is the result of executing all its basic blocks. Therefore, retrieving two equivalent paths is an initiation to further match their nodes. The longer the path is, the more matching pairs would be acquired.

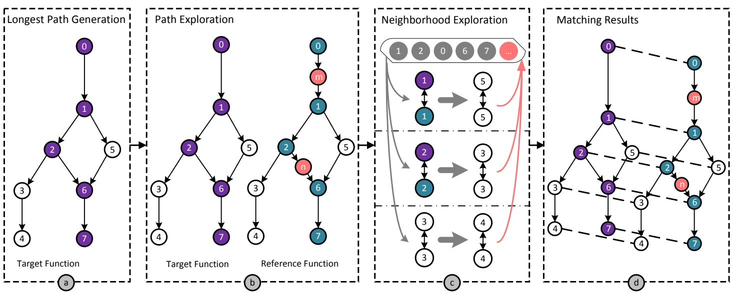


Fig. 6: Fuzzy matching

Second, the reference function is explored to find the best match for the longest path in the target function. Inspired by [36, 44], a breadth-first search combined with longest common subsequence (LCS) method of dynamic programming [21] is executed. In order to satisfy the requirements of the LCS algorithm, since any path is a sequence of basic blocks, each basic block is treated as a letter. Two basic blocks are compared based on their instructions, and a similarity score (which will be discussed later) is returned. Therefore, all the possible paths in the reference function are explored and the one with the highest similarity score is returned as the best matched path (including basic blocks pairs) [36]. Additionally, we put all the obtained matching basic blocks pairs in a priority queue. As an example, the best match for the given longest path with a reference function is highlighted in Figure 6 part *b*.

Finally, we further perform neighbourhood exploration and leverage Hungarian algorithm in both the target and reference functions to improve and extend the mapping. Since all the mapping basic block pairs are obtained as the result of path exploration, we explore the neighbours of the most similar basic block pairs (priority queue shown in Figure 6 part *c*) to initiate the search and find more matched pairs for their successors and predecessors by considering the in-degrees and out-degrees and leveraging Hungarian algorithm. If there is a new match, we put the paired match in the priority queue to explore their neighbours later on. We continue the same algorithm for the rest of nodes until the priority queue is empty. The outcome of neighbourhood exploration is the basic block matching pairs in the control flow graph (Figure 6 part *d*) and the corresponding similarity scores. To obtain the final similarity score between the f_T and f_r functions with n_T and n_r number of basic blocks, respectively, we apply the following formula:

$$similarity\ (f_T, f_r) = \frac{2 \times \sum_{i=1}^{k} WJ(S,T)}{n_T + n_r} \tag{2}$$

where k is the number of matched basic blocks between functions f_T and f_r, and $WJ(S,T)$ returns the similarity score between the matching basic block pairs. Moreover, BinArm provides all the differences between two functions at instruction level, basic blocks level and function level.

Basic Block Matching. For basic block matching, we could adopt the LCS method of dynamic programming on the instructions of two basic blocks as in [36]. However, the accuracy of this approach might be affected by *instruction reordering* and *instruction substitutions* [36]. Moreover, the time complexity of the LCS algorithm is $O(mn)$, where m and n represent the number of instructions in the two basic blocks. Consequently, to accurately and efficiently perform basic block matching, we use the weighted Jaccard similarity [37] between the two basic blocks. Let two sets of S and T contain the mnemonic frequencies of the two basic blocks, with n and m number of elements in each blocks. The weighted Jaccard similarity (WJ) between the two vectors is calculated as follows:

$$WJ(S,T) = \frac{\sum_{k=1}^{N} \min(S_k \cap T_k)}{\sum_{k=1}^{N} \max(S_k \cup T_k)},\ N = \max\{m, n\}$$

The usage of WJ similarity together with instruction grouping could overcome *instruction reordering* and some *instruction substitutions*. Moreover, the time complexity of the WJ similarity is of order $O(N)$.

5 Evaluation

This section details our experiments and analysis.

5.1 Experimental Setup

All of our experiments are conducted on machines running Windows 7 and Ubuntu 15.04 with Intel Xenon $E5$ 2.4 GHz CPU and 16GB RAM. BinArm is written in C++ and utilizes a *Cassandra* database [6] to store all the functions along with their features. *Vagrant* [16] is used to create a specialized environment used for firmware reverse engineering as well as library cross compilation for the *ARM* architecture. The utilized cross compiler is *gcc-arm-linux-gnueabi* version 4.7.3 using the debug flag (*-g*), the static flag (*-static*), and all compatible optimization flags (*-O0, -O1, -O2, -O3*). The symbol names are preserved during the compilation process for metric validation. A custom Python script is used in tandem with IDA Pro [10] to extract function CFGs in the desired JSON format. *Docker* [9] is used to create a containerized version of the CVE database and its associated search tools [8].

Dataset. The experiments are performed on different datasets, which are explicitly indicated in each section. In order to evaluate the scalability of BinArm, a large quantity of firmware images (IED and non-IED firmware images) are collected from the wild, 5,756 of which were successfully disassembled to construct our *General Dataset*.

Evaluation Metrics. To evaluate the accuracy of BinArm, we use the $F_1 = 2 \times \frac{P \times R}{P+R}$ measure, where $P = \frac{TP}{TP+FP}$ is the precision, and $R = \frac{TP}{TP+FN}$ is the recall. In addition, TP is the number of relevant functions that are correctly retrieved; FP represents the number of irrelevant functions that are incorrectly detected; and FN indicates the number of relevant functions that are not detected, and TN represents the number of irrelevant functions that are not detected. Total accuracy (TA) is measured as $TA = \frac{TP+TN}{TP+TN+FP+FN}$.

Time Measurement. The execution time for function indexing is measured by adding the time required for each step, including feature extraction and function indexing. The search time includes time required for feature extraction and function discovery. The time taken to disassemble the binaries using IDA Pro is excluded, where it takes on the order of seconds on average to disassemble a binary file and can be distributed over all functions in a binary file.

5.2 Function Identification Accuracy

We evaluate the accuracy of BinArm by examining a randomly selected set of binaries from our *Vulnerability Database*, where the source code and the symbol names are provided in order to validate the results. We randomly select

10% of libraries from *Vulnerability Database* as target libraries, and match them against 90% remaining libraries in our repository. The average accuracy results are summarized in Table 4. As can be seen, the average of total accuracy is 0.92. According to our experiments, the results are affected due to different versions and the degree of changes in the new versions. Since the libraries are randomly selected, in some cases the differences between versions are relatively high that cause a drop in the accuracy.

Table 4: Average accuracy results

Project	`glibc`	`libcurl`	`libxml2`	`lighttpd`	`ntp`	`openssh`	`openssl`	`postgresql`	`zlib`	**Average**
Total Accuracy	0.96	0.93	0.89	0.92	0.87	0.89	0.93	0.98	0.89	0.92

5.3 Efficiency

In this section, we conduct experiments to measure the efficiency of BinArm for function matching. To this end, we test the 5,103 vulnerable functions against all functions in our *Vulnerability Database* and Netgear ReadyNAS v6.1.6 firmware separately, and measure the search time for each function. The obtained results are reported in Figure 7, where the x-axis represents the percentage of number of functions, and the y-axis shows the cumulative distribution function (CDFs) of search time. The average searching times per function for each scenario are 0.01 seconds and 0.008 seconds, respectively. It is worth noting that the search time of BinArm is firmly related to the CFG complexity of target function. If the target function has a large value of `graph_energy`, the search time would be higher. However, search time of a small function against a very complex CFG would not be costly, since the complex functions are deemed dissimilar in the shape-based detection stage and filtered out, and no heavy graph matching would be performed in the next detection stages.

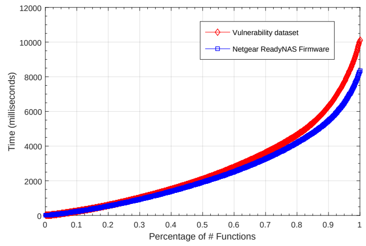


Fig. 7: CDF of vulnerable function search time

5.4 Comparison

Indexing Time Comparison. In order to compare the indexing time of BinArm with the state-of-the-art discovRE [28], Genius [30], and Multi-MH [50] approaches, we choose the Netgear ReadyNAS v6.1.6 [13] firmware image. The reasons of this choice are threefold: (i) the firmware is publicly available and is based on the ARM architecture; (ii) all the aforesaid works have measured the indexing time of Netgear ReadyNAS based on their techniques; and (iii) the hardware specifications of the machines of conducted experiments are provided. Altogether these facilitate comparison. We index ReadyNAS in our database and record the indexing time. Table 5 illustrates the preparation time along with the

Table 5: Baseline comparison on indexing time of `ReadyNAS v6.1.6`

	Multi-MH [50]	**Genius** [30]	**BinArm**	**discovRE** [28]
Time (Minutes)	$5,475$	89.7	78.65	54.1[a]
Hardware Specification	Intel Core $i7-2640M$ at 2.8GHz 8GB DDR3-RAM	24 Cores at 2.8 GHz 65GB RAM	Intel Xenon $E5-2630v3$ at 2.4 GHz 16GB RAM	Intel Core $i7-2720QM$ at 2.20 GHz 8GB DDR3 RAM

[a] discovRE only considers the CFG extraction time, while BinArm extracts additional features, such as the weighted paths.

hardware specifications that are reported by the aforementioned approaches, as well as those of BinArm. As taking machines computational capacity into account, BinArm is more efficient with respect to indexing time when compared to aforesaid approaches with the exception of discovRE. The reason is that discovRE only considers CFG extraction time, while BinArm extracts additional features, such as the weighted paths. Nevertheless, the evaluation performed by [30] demonstrate discovRE's inaccuracy in large scale setup.

Search Time Comparison. We further compare the search time of our prototype system with that of BinSequence [36]. The reason for this comparison is to verify the efficiency of the first two stages of detection prior to the third stage of fuzzy matching, as BinSequence employs fuzzy matching approach after a pre-filtering process. In this experiment, we compare three different versions of `zlib` library (v1.2.5, v1.2.6, v1.2.7) with their next version using BinArm with the same setup performed in BinSequence. For example, we test `zlib` v1.2.5 against its successive version `zlib` v1.2.6 together with two million noise functions in the database. We collect the search time for each scenario, and obtain the average time of 0.0002 seconds per function as reported in Table 8. On the other hand, the average of optimal search times for these three scenarios provided by BinSequence [36] is 0.909 seconds per function. These results confirm that BinArm is three orders of magnitude faster than BinSequence.

Fig. 8: Baseline comparison on search time (seconds) per function

zlib Version	**BinArm**	**BinSequence** [36]
1.2.5	0.00057	0.897
1.2.6	0.00016	0.913
1.2.7	0.00009	0.918
Average	**0.00027**	**0.909**

Qualitative Comparison with Gemini. Gemini [60] is one of the latest iterations in code similarity detection in binaries, which extracts attributed control flow graphs and feeds them into a siamese neural network. Since the tool is not publicly available in order to perform a direct comparison, a qualitative comparison is performed as follows. (i) The required training time of Gemini, which is performed on a powerful server with two CPUs and one GPU card, is significant compared to BinArm. (ii) The time required to constantly retrain the neural network and re-generate the embeddings is a major disadvantage in a real-world scenario. As such, BinArm greatly outperforms Gemini with respect to the indexing of new vulnerable functions into the system. (iii) Gemini has a total of

154 vulnerable functions and presents a use case that employs two of them. In contrast, BinArm *Vulnerability Database* contains 235 vulnerable functions, all of which are used for vulnerability identification. (iv) Gemini solely relies on a few basic features and the use of a siamese neural network to perform the comparison. Such feature choices are reflected through the reported vulnerability identification accuracy of about 82% [60], whereas BinArm's much richer collection of features and the rigorous feature selection process help to obtain a 92% accuracy. This is partially due to the fact that BinArm takes into account a much broader scope of information relative to a given function. For example, by only counting the number of arithmetic instructions, Gemini barely takes into account function semantics. In contrast, BinArm extracts far more semantics from a function through its branch-based detection, which each basic block instruction probability densities are leveraged, and its fuzzy matching, which the features at different levels are taken into account.

5.5 Detecting Vulnerabilities in Real Firmware

In this section, we demonstrate BinArm's capability to facilitate the vulnerability identification process in real-world IED firmware. We randomly select five firmware images from our *Firmware Database* and compare them to all vulnerable functions in our *Vulnerability Database*. Each resulting function pair is ranked by similarity scores. We consider a candidate as a potential match, if the matching score is higher than 80%. We successfully identify 93 potential CVEs in the randomly selected firmware images, 75 of which are confirmed by our manual analysis.

Table 6: Identifying CVEs in real-world firmware images

Firmware	CVE	Score	Firmware	CVE	Score
`NI PMU1_0_11`	CVE-2016-6303	1.00	`Schneider Link150`	CVE-2015-0208	0.68
	CVE-2014-8176	1.00	`Schneider M251`	CVE-2014-2669	0.65
	CVE-2014-6040	0.92	`ReadyNAS v6.1.6`	CVE-2015-7497	0.98
	CVE-2016-7167	0.91		CVE-2014-2669	0.97
	CVE-2015-0288	0.91		CVE-2015-7941	0.95
`Honeywell.RTUR150`	CVE-2016-0701	1.00		CVE-2014-6040	0.93
	CVE-2016-2105	0.99		CVE-2010-1633	0.93
	CVE-2010-1633	0.94		CVE-2014-0160	0.92
	CVE-2016-6303	0.94		CVE-2015-0288	0.91
	CVE-2015-0287	0.92		CVE-2014-0160	0.86

Due to lack of space, a subset of obtained results has been presented in Table 6. As can be seen, BinArm is able to successfully identify different vulnerabilities in the NI PMU1_0_11, Honeywell.RTUR150, and ReadyNAS v6.1.6 firmware images. For instance, it is able to identify CVE-2016-7167 (critical heap-based buffer overflow vulnerability [7]) in the NI PMU1_0_11 firmware as the first rank with 0.91 similarity score. The obtained matching results of vulnerable function `X509_to_X509_REQ` and the matched one in NI PMU1_0_11 firmware are depicted in Figures 9. This output verifies the results, and illustrates BinARM's capability that provides in-depth mapping results for the verification purpose. Additionally, our experiments demonstrate that BinArm can identify the CVE-2014-0160 (Heartbleed vulnerability) and CVE-2014-3566 (POODLE vulnerability) in ReadyNAS firmware (as demonstrated in the state-of-the-art approaches [28, 30]) in less than 0.5 ms. The results confirm the capability of BinArm to be applied in real-world scenarios to perform vulnerability analysis on the IED firmware embedded in the smart grid.

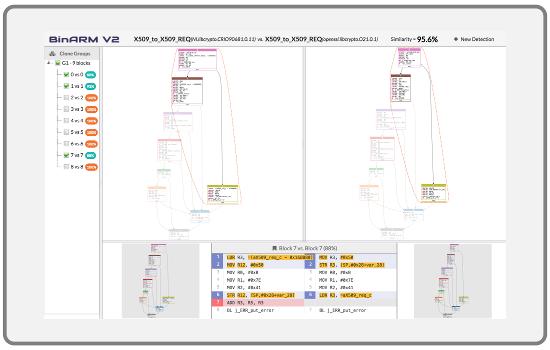


Fig. 9: Results of bug Search in NI PMU1_0_11 firmware

5.6 Impact of Multiple Detection Stages

In order to study the impact of proposed multi-stage detection engine, we employ four experiments by enabling and disabling shape-based and branch-based detectors (we always keep the fuzzy matching-based detector enabled), and measure both the accuracy and efficiency of BinArm on *Vulnerability* database. To this end, we perform the test on a randomly selected projects with different versions and optimization settings. As can be seen in Table 7, the total accuracy remains the same as it is not affected by any of the prior detection stages. On the other hand, the proposed multi-stage detection improves the efficiency of BinArm.

Table 7: Impact of Multiple Detection Stages

Shape-based	**Branch-based**	**Accuracy**	**Time(s)**
True	True	0.929	626.72
True	False	0.928	3649.80
False	True	0.925	44823.34
False	False	0.924	50671.66

Note: The Fuzzy-based detector is always enabled.

5.7 Impact of Parameters

In this subsection, we provide the impact of λ and γ on BinArm accuracy. We perform experiments by (i) disabling the branch-based detector, and incrementing the value of λ by 5 starting from 5; (ii) disabling the shape-based detector and incrementing the value of γ by 5 each time, starting from 30%. We randomly select 10% of libraries from our *Vulnerability Dataset* as the test set, and perform the matching against remaining libraries in our dataset and record the accuracy. The experimental results illustrated in Figure 10 and Figure 11 demonstrate that the obtained values of $\lambda = 26.45$ and $\gamma = 50\%$ return the highest accuracy among other values.

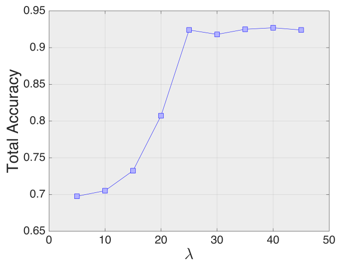


Fig. 10: Impact of λ

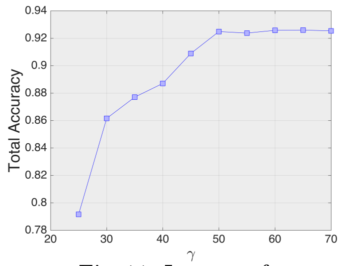


Fig. 11: Impact of γ

5.8 Scalability Study

We further investigate the time required for both indexing and retrieving matched functions to demonstrate BinArm capability to handle firmware analysis at a large scale. To this end, we randomly index one million functions from *General Dataset*, and collect the indexing time per function. Figure 12 depicts the CDF of the preparation time for the randomly selected functions. As shown, most of the functions are indexed in less than 0.1 second, where the median indexing time is 0.008 seconds, and it takes 0.02 seconds on average to index a function.

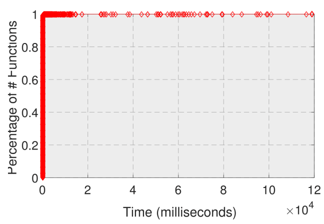


Fig. 12: CDF of indexing time for 1 million functions

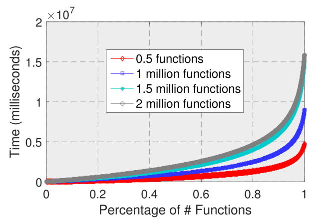


Fig. 13: CDF of search time against incrementing reference functions

Moreover, we perform several scalability benchmarks, each utilizes a randomly selected set of 10,000 target functions. For each evaluation, we employ a randomly selected set of reference functions, where its size increases in increments of 0.5 up to 2 million, as plotted in Figure 13.

6 Related Work

We briefly describe most recent existing works to identify known vulnerable functions in program binaries. BinDiff [26] performs graph isomorphism on function pairs of two binaries in the cross architecture setting. However, it is not designed to be applied on large scale datasets. Rendezvous [39] performs function matching based on the mnemonics, n-grams, CFGs and data constants extracted from functions. Nevertheless, this approach is sensitive to structural changes and instruction reordering. TRACY [23] employs longest common subsequence algorithm to align two tracelets obtained from decomposed CFGs. However, it is suitable for functions with more than 100 basic blocks [23]. BinSequence [36] compares two functions using longest common subsequence and neighbourhood exploration. However, its accuracy drops due to the effects of code transformation [36]. Moreover, the proposed MinHash-based filtering is not efficient for large and complex functions.

Some cross-architecture bug search approaches have been proposed. For instance, Multi-MH [50] finds similar code by capturing the input and output variables at basic block level. However, finding semantic similarities is performed by MinHash, which is slow to be applicable to large code base. discovRE [28] applies maximum common subgraph isomorphism on the CFGs to find similar functions, whereas the utilized pre-filtering to speed up the subgraph isomorphism causes significant reduction in accuracy [30]. Genius [30] generates attributed control flow graphs, where each basic block is labelled with statistical and structural features, and then converts them into embeddings using locality sensitive hashing (LSH). However, graph embedding and distance matrix is expensive [60], and changes in the CFG structure affect its accuracy [30]. Most recently, a neural network-based approach called Gemini [60] computes numeric vectors based on the CFGs and addresses the efficiency issue of Genius. We compare BinArm with the aforementioned proposals in Table 8.

Table 8: Comparing existing solutions with BinArm

PROPOSALS	Feature				Feature Level			Architecture			Compiler			
	Syntactic	Semantic	Structural	Statistical	Instruction	Basic Block	Function	x86-64	ARM	MIPS	VS	GCC	ICC	Clang
BinDiff [26]		•	•			•	•	•			•	•		
Rendezvous [39]	•	•	•		•		•	•				•		
TRACY [23]		•	•			•	•	•				•		
BinSequence [36]	•	•	•		•	•	•	•				•		
Multi-MH [50]		•				•		•	•	•		•		•
discovRE [28]			•	•	•	•		•	•	•	•			
Genius [30]			•	•	•	•		•	•	•		•		•
BinShape [54]	•	•	•	•			•	•			•	•		
Gemini [60]			•	•	•	•		•	•	•		•		•
BinArm	•	•	•	•	•	•	•		•			•		

Note: Symbol (•) indicates that system supports the corresponding feature, otherwise it is empty.

All of the aforesaid approaches employ static analysis, while some dynamic analysis techniques have been proposed. For instance, BLEX [27] executes functions for several calling contexts and deems functions with the same side effects as similar. However, dynamic analysis approaches are often computationally expensive, and are difficult for firmware images [28].

7 Conclusion

The heavy use of intelligent electronic devices (IEDs) in industrial control systems for critical infrastructures, such as smart grid, increases its importance criticality. Recent studies highlight the security evaluation of firmware images as the foremost step to ensure security and functionality in those critical infrastructures. In this paper, we presented BinArm, a scalable and efficient vulnerability detection technique for the IED firmware. To this end, we proposed two substantial databases of smart grid firmware, and relevant vulnerabilities. We then introduced a multi-stage detection engine that could leverage this data to identify vulnerable functions in IED firmware pertaining to the smart grid. Extensive experiments demonstrated that BinArm could accurately and efficiently perform the vulnerability identification process. This was further ramified by its evaluation on real-world IED firmware images and its success in identifying 93 potentially vulnerable functions and having confirmed 75 of them. However, BinArm has the following limitations: (i) *Function Inlining.* We do not currently support function inlining. However, this problem can be circumvented by leveraging data flow analysis. (ii) *Multiple Architecture.* Our system deals with only ARM hardware architecture, since most of IEDs embedded in ICSs are based on ARM processors. An intermediate representation could be leveraged to support multiple architectures. (iii) *Type Inference.* We do not consider type inference in our features. However, type information is important to mitigate some sort of vulnerabilities [17, 58]. (iv) *Runtime Vulnerability Detection.* The proposed detection approach fails to detect runtime data-oriented exploits, due to the lack of runtime execution semantics checking [20]. Therefore, proposing a hybrid approach including dynamic analysis could overcomes this limitation.

References

1. ICS-CERT: Critical Infrastructure Sectors. `https://www.dhs.gov/critical-infrastructure-sectors`.
2. IEC 61850 - Communication networks and systems for power utility automation. `https://webstore.iec.ch/publication/6028`. Accessed: 2018.
3. WIN32/INDUSTROYER A new threat for industrial control systems. `https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32\_Industroyer.pdf`.
4. NIST/SEMATECH e-Handbook of Statistical Methods, 2015. `http://www.itl.nist.gov/div898/handbook/`.
5. ARM Instruction Reference, 2017. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/CIHEDHIF.html`.
6. Cassandra DB, 2017. `https://cassandra.apache.org/`.
7. Common Vulnerabilities and Exposures, 2017. `https://nvd.nist.gov/`.
8. cve-search-docker, 2017. `https://github.com/leojcollard/cve-search-docker`.
9. Docker, 2017. `https://www.docker.com/`.
10. IDA Pro, 2017. `https://www.hex-rays.com/products/ida/`.
11. NI Copyright, 2017. `http://zone.ni.com/reference/en-XX/help/374498F-01/insightcm/bp\_copyright/`.
12. NI PMU1_0_11.lvappimg, 2017. `http://digital.ni.com/public.nsf/allkb/5391E8424944D0BC86257E45000B025C`.
13. ReadyNAS Firmware Image v6.1.6, 2017. `http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip`.
14. Scrapy, 2017. `https://scrapy.org/`.

15. Security Intelligence, 2017. `https://securityintelligence.com/attacks-targeting-industrial-control-systems-ics-up-110-percent/`.
16. Vagrant, 2017. https://www.vagrantup.com/.
17. Caballero, J., and Lin, Z. Type inference on executables. *ACM Computing Surveys (CSUR) 48*, 4 (2016), 65.
18. Chen, B., Dong, X., Bai, G., Jauhar, S., and Cheng, Y. Secure and efficient software-based attestation for industrial control devices with arm processors. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (2017), ACM, pp. 425–436.
19. Chen, D. D., Egele, M., Woo, M., and Brumley, D. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS* (2016).
20. Cheng, L., Tian, K., and Yao, D. D. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks.
21. Cormen, T. H. *Introduction to algorithms.* MIT press, 2009.
22. Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., and Antipolis, S. A large-scale analysis of the security of embedded firmwares. In *USENIX Security* (2014).
23. David, Y., and Yahav, E. Tracelet-based code search in executables. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 349–360.
24. Davidson, D., Moench, B., Ristenpart, T., and Jha, S. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security* (2013), pp. 463–478.
25. Dimitriadou, E., Dolničar, S., and Weingessel, A. An examination of indexes for determining the number of clusters in binary data sets. *Psychometrika 67*, 1 (2002), 137–159.
26. Dullien, T., and Rolles, R. Graph-based comparison of executable objects (english version). *SSTIC 5* (2005), 1–3.
27. Egele, M., Woo, M., Chapman, P., and Brumley, D. Blanket execution: Dynamic similarity testing for program binaries and components. In *Usenix Security* (2014), pp. 303–317.
28. Eschweiler, S., Yakdan, K., and Gerhards-Padilla, E. discovRe: Efficient cross-architecture identification of bugs in binary code. In *NDSS* (2016).
29. Falliere, N., Murchu, L. O., and Chien, E. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response 5*, 6 (2011).
30. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., and Yin, H. Scalable graph-based bug search for firmware images. In *CCS* (2016), ACM.
31. Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. Structural detection of android malware using embedded call graphs. In *AISec* (2013), ACM.
32. Griffin, C. Graph theory: Penn state math 485 lecture notes. `http://www.personal.psu.edu/cxg286/Math485.pdf`, 2011-2012.
33. Groarke, D. G. R. The networked grid 150: The end-to-end smart grid vendor ecosystem report and rankings 2013. `https://www.greentechmedia.com/research/report/the-networked-grid-150-report-and-rankings-2013`.
34. Han, J., Pei, J., and Kamber, M. *Data mining: concepts and techniques.* Elsevier, 2011.
35. Hido, S., and Kashima, H. A linear-time graph kernel. In *ICDM* (2009).
36. Huang, H., Youssef, A. M., and Debbabi, M. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *ASIACCS* (2017), ACM.
37. Ioffe, S. Improved consistent sampling, weighted minhash and l1 sketching. In *ICDM* (2010).

38. J. Leeds, D. G. R. The networked grid 150: The end-to-end smart grid vendor ecosystem report and rankings. https://www.greentechmedia.com/research/report/the-networked-grid-150.
39. Khoo, W. M., Mycroft, A., and Anderson, R. Rendezvous: A search engine for binary code. In *MSR* (2013).
40. Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. Static disassembly of obfuscated binaries. In *USENIX security* (2004).
41. Kwon, Y., Kim, H. K., Koumadi, K. M., Lim, Y. H., and Lim, J. I. Automated vulnerability analysis technique for smart grid infrastructure. In *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017 IEEE* (2017), IEEE, pp. 1–5.
42. Langner, R. Stuxnet: Dissecting a cyberwarfare weapon. In *IEEE SP* (2011).
43. Liu, M., Zhang, Y., Li, J., Shu, J., and Gu, D. Security analysis of vendor customized code in firmware of embedded device. In *International Conference on Security and Privacy in Communication Systems* (2016), Springer, pp. 722–739.
44. Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *ACM SIGSOFT* (2014).
45. Mackiewicz, R. Overview of iec 61850 and benefits. In *Power Systems Conference and Exposition, 2006. PSCE'06. 2006 IEEE PES* (2006), IEEE, pp. 623–630.
46. Nazario, J. Blackenergy ddos bot analysis. *Arbor Networks* (2007).
47. Neichin, G., Cheng, D., Haji, S., Gould, J., Mukerji, D., and Hague, D. 2010 US smart grid vendor ecosystem.
48. Oliver, J., Cheng, C., and Chen, Y. Tlsh–a locality sensitive hash. In *CTC* (2013), IEEE.
49. Peng, H., Long, F., and Ding, C. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE TPAMI* (2005).
50. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., and Holz, T. Cross-architecture bug search in binary executables. In *IEEE SP* (2015).
51. Rad, B. B., Masrom, M., and Ibrahim, S. Opcodes histogram for classifying metamorphic portable executables malware. In *ICEEE* (2012).
52. Rieck, K., Holz, T., Willems, C., Düssel, P., and Laskov, P. Learning and classification of malware behavior. In *DIMVA* (2008).
53. Series, I. Business blackout. https://www.lloyds.com/~/media/files/news-and-insight/risk-insight/2015/business-blackout/business-blackout20150708.pdf.
54. Shirani, P., Wang, L., and Debbabi, M. BinShape: Scalable and robust binary library function identification using function shape. In *DIMVA* (2017).
55. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS* (2015).
56. Shu, X., Yao, D., and Ramakrishnan, N. Unearthing stealthy program attacks buried in extremely long execution paths. In *CCS* (2015), ACM.
57. Shwartz, O., Mathov, Y., Bohadana, M., Elovici, Y., and Oren, Y. Opening pandora's box: Effective techniques for reverse engineering iot devices. In *International Conference on Smart Card Research and Advanced Applications* (2017), Springer, pp. 1–21.
58. Slowinska, A., Stancescu, T., and Bos, H. Body armor for binaries: Preventing buffer overflows without recompilation. In *USENIX Annual Technical Conference* (2012), pp. 125–137.

59. Wang, T., Wei, T., Lin, Z., and Zou, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS* (2009).
60. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., and Song, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 363–376.
61. Zaddach, J., Bruno, L., Francillon, A., and Balzarotti, D. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS* (2014).
62. Zaddach, J., and Costin, A. Embedded devices security and firmware reverse engineering. *Black-Hat USA* (2013).
63. Zheng, Y., Ott, W., Gupta, C., and Graur, D. A scale-free method for testing the proportionality of branch lengths between two phylogenetic trees. *arXiv preprint arXiv:1503.04120* (2015).
64. Zhu, R., Zhang, B., Mao, J., Zhang, Q., and Tan, Y.-a. A methodology for determining the image base of arm-based industrial control system firmware. *International Journal of Critical Infrastructure Protection 16* (2017), 26–35.