Hybrid Firmware Analysis for Known Mobile and IoT Security Vulnerabilities

Pengfei Sun†, Luis Garcia*, Gabriel Salles-Loustau† and Saman Zonouz†
† Electrical and Computer Engineering, *Electrical and Computer Engineering
†Rutgers University, *University of California, Los Angeles
{*pengfei.sun, gabriel.sallesloustau, saman.zonouz*}*@rutgers.edu*, {*garcialuis*}*@ucla.edu*

***Abstract*—Mobile and IoT operating systems–and their ensuing software updates–are usually distributed as binary files. Given that these binary files are commonly closed source, users or businesses who want to assess the security of the software need to rely on reverse engineering. Further, verifying the correct application of the latest software patches in a given binary is an open problem. The regular application of software patches is a central pillar for improving mobile and IoT device security. This requires developers, integrators, and vendors to propagate patches to all affected devices in a timely and coordinated fashion. In practice, vendors follow different and sometimes improper security update agendas for both mobile and IoT products. Moreover, previous studies revealed the existence of a *hidden patch gap*: several vendors falsely reported that they patched vulnerabilities. Therefore, techniques to verify whether vulnerabilities have been patched or not in a given binary are essential. Deep learning approaches have shown to be promising for static binary analyses with respect to inferring binary similarity as well as vulnerability detection. However, these approaches fail to capture the dynamic behavior of these systems, and, as a result, they may inundate the analysis with false positives when performing vulnerability discovery in the wild. In particular, they cannot capture the fine-grained characteristics necessary to distinguish whether a vulnerability has been patched or not.**

In this paper, we present PATCHECKO, a vulnerability and patch presence detection framework for executable binaries. PATCHECKO relies on a hybrid, cross-platform binary code similarity analysis that combines deep learning-based static binary analysis with dynamic binary analysis. PATCHECKO does not require access to the source code of the target binary nor that of vulnerable functions. We evaluate PATCHECKO on the most recent Google Pixel 2 smartphone and the Android Things IoT firmware images, within which 25 known CVE vulnerabilities have been previously reported and patched. Our deep learning model shows a vulnerability detection accuracy of over 93%. We further prune the candidates found by the deep learning stage–which includes false positives–via dynamic binary analysis. Consequently, PATCHECKO successfully identifies the correct matches among the candidate functions in the top 3 ranked outcomes 100% of the time. Furthermore, PATCHECKO's differential engine distinguishes between functions that are still vulnerable and those that are patched with an accuracy of 96%.

I. INTRODUCTION

The number of discovered software vulnerabilities and the rate at which we discover them increase steadily every year. The number of new vulnerability reports submitted to the Common Vulnerabilities and Exposures (CVE) database was approximately $4,600$ in 2010, $6,500$ in 2016, and doubled in 2017 with over $14,700$ reports [30]. In parallel, the increasing ubiquity of mobile and IoT devices (Gartner forecasts that 20.4 billion IoT devices will be in use worldwide by 2020 [40]) makes them a target of choice for vulnerability research and exploitation. Further, it is common practice for both customers and businesses to rely on commercial off-the-shelf binaries in their products or for their activities. These external products often require a vetting step, including security assertion of the product's software, e.g., blackbox penetration testing. When not done properly, such use or integration of IoT devices can lead to security issues [10]. Fortunately for the penetration testers, mobile and IoT vendors often reuse open source code and adapt them to their products. Common targets for penetration testers are binary files such as cryptographic libraries, media libraries, and parsers that are regularly updated upon vulnerability discovery. Unfortunately, the source code for these libraries in mobile and IoT devices is not always easily accessible, and ensuring that their software is up to date is an open problem.

Generally, patch management for both IoT and mobile devices is a challenge for heterogeneous ecosystems. A 2018 Federal Trade Commission report [11] mentioned that although an ecosystem's diversity provides extensive consumer choice, it also contributes to security update complexity and inconsistency. Software patches must go through many intermediaries from the software developers, to the software integrators, and onto the vendors before getting pushed to the end devices [1].

Two problems arise from this long patch chain. First, this long list of intermediaries tends to delay the propagation of patches to the end device. Duo labs found that only 25 percent of mobile devices were operated on an up-to-date patch level in 2016 [26]. Second, vendors do not always accurately report whether a vulnerability has been patched or not (*hidden patch gaps*), especially in the context of mobile and IoT devices. A study showed that 80.4% of vendor-issued firmware is released with multiple known vulnerabilities, and many recently released firmware updates contain vulnerabilities in third-party libraries that have been known for over eight years [13]. Another study of Android phones [27] found that some vendors regularly miss patches, leaving parts of the ecosystem exposed to the underlying risks. Hidden patch gaps not only leave a large set of devices vulnerable, but with the pervasiveness of software code reuse, such vulnerabilities can quickly propagate as developers may copy over existing vulnerabilities [12].

Accordingly, identifying vulnerable binaries and patch status is a critical challenge for end users. PATCHECKO solves this problem via a two-step hybrid approach that combines a lightweight whole firmware static analysis followed by an accurate dynamic analysis that refines the static analysis results.

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00053

Known vulnerability discovery via deep learning. Obtaining the set–or at least a superset–of candidate vulnerabilities for a given binary is an explored problem without a satisfactory answer. Recently, researchers have started to tackle the cross-platform binary similarity checking to detect known vulnerabilities [32], [17], [16], [41]. These efforts try to identify the functions, if any, in the target firmware that "look like" one of the functions in a previously populated database of functions with known vulnerabilities. They propose to extract various robust, platform-independent features directly from binary code for each node in the control flow graph that represents a function. Other approaches have focused on binary similarity detection where a graph matching algorithm is used to check whether two functions' control flow graph representations are similar [32], [17], [16]. Further, deep learning in the context of Natural Language Processing (NLP) can also replace manually selected features [43], [14].

Prior efforts have shown that deep learning approaches can be used for binary analysis to detect vulnerabilities[38], [41], [9]. The most recent approach [41] has a training performance of 0.971 Area Under the Curve (AUC) and detection accuracy of over 80%. However, despite this performance, assuming the target binary has around 3000+ functions, there is still a large number of candidate functions (600+) that need to be explored manually for confirmation after the analysis. It has been shown the candidate functions can be pruned given access to a binary's symbol table [42]. However, for stripped commercial-off-the-shelf (COTS) binaries, their solution can only provide a very large set of candidate functions (mostly false positives). Accordingly, further measures are necessary to prune the candidate functions to identify and report only the true positives (the functions with actual vulnerabilities). PATCHECKO uses the target binary static analysis results (*static features*) to conduct this first stage.

Candidate function pruning via dynamic analysis. PATCHECKO prunes the set of candidate functions from the deep learning-based approach with dynamic analysis results (*dynamic features*) to get rid of the false positives. The static analysis removes the bulk of improbable candidates and returns a small subset of functions, which enables PATCHECKO to consider more resource-expensive dynamic analysis techniques on a smaller set of target functions. Prior work [17], [41] prioritized speed at the expense of accuracy due to scalability concerns–focusing only on heuristic or static features of basic blocks and functions. Compared to [15], PATCHECKO's hybrid approach not only speeds up the vulnerability function matching process, but also provides higher accuracy via removing false positives.

This initial framework allows us to develop a new training model generation method that uses a default policy to pretrain a task-independent graph embedding network. We then use this method to generate a large-scale dataset using binary functions compiled from the same source code but for different platforms with different compiler optimization levels. We then built a vulnerability database that includes 1,382 vulnerabilities for mobile/IoT firmware.

However, the ultimate goal of our solution is not to only find similar vulnerability functions. The final goal is to ensure whether the vulnerability is still in the target firmware or if it has been patched.

Missing patch detection. Prior work has already developed precise patch presence tests [42]. However, this solution only works with access to the source code for both the vulnerable and patched function source code. Also, because this solution relies on binary similarity-based approaches to locate target functions, it suffers from the aforementioned high false positive rate for candidate functions. Our solution works directly with stripped COTS binaries and does not require access to the source code while significantly pruning false positives.

Ultimately, this paper presents PATCHECKO: a framework that integrates deep-learning for binary similarity-checking with dynamic analysis to discover known vulnerabilities as well as to test for patch presence. Our evaluation demonstrates that PATCHECKO significantly outperforms the state-of-the-art approaches with respect to both accuracy and efficiency.

Contributions. We summarize our contributions as follows:

- We propose an efficient firmware vulnerability and patch presence detection framework that leverages deep learning and dynamic binary analysis techniques to achieve high accuracy and performance for known vulnerability discoveries in stripped firmware binaries without source code access.
- We propose a fine-grained binary comparison algorithm to distinguish accurately between patched and unpatched versions of the same function's binaries. Our solution currently works cross-platform–supporting ARM and X86 architectures. The selected relevant features for the comparison enable our solution to pinpoint the unpatched functions with a very low false positive rate.
- We evaluate PATCHECKO on 25 CVE vulnerabilities for 100 different Android firmware libraries across 4 different architectures. Our results are very promising for practical deployment in real settings. With most of PATCHECKO's prototype being fully automated, its dynamic analysis module correctly identified and pruned the false positives from the deep learning classification outcomes. The results were later processed, and the unpatched functions were separated from the functions with already-patched vulnerabilities.

II. OVERVIEW

We introduce the vulnerability function similarity problem and challenges in II-A and then present our solution in II-B.

A. Threat Model and Challenges

In this paper, we consider the problem of searching for known vulnerabilities in stripped COTS mobile/IoT binaries. We assume that we do not have access to the source code. We also assume that the binary is not packed or obfuscated and that the binary is compiled from a high-level procedural programming language, i.e., a language that has the notion of functions. While handling packed code is important, it poses unique challenges, which are out of scope for this paper. Considering these assumptions, we identify the following challenges that arise in the domain of mobile/IoT platforms.

Heterogeneous binary compilation. Mobile/IoT platforms typically consist of heterogeneous distributions of hardware that may share common software vulnerabilities. As such, we explicitly consider cases where different cross-platform

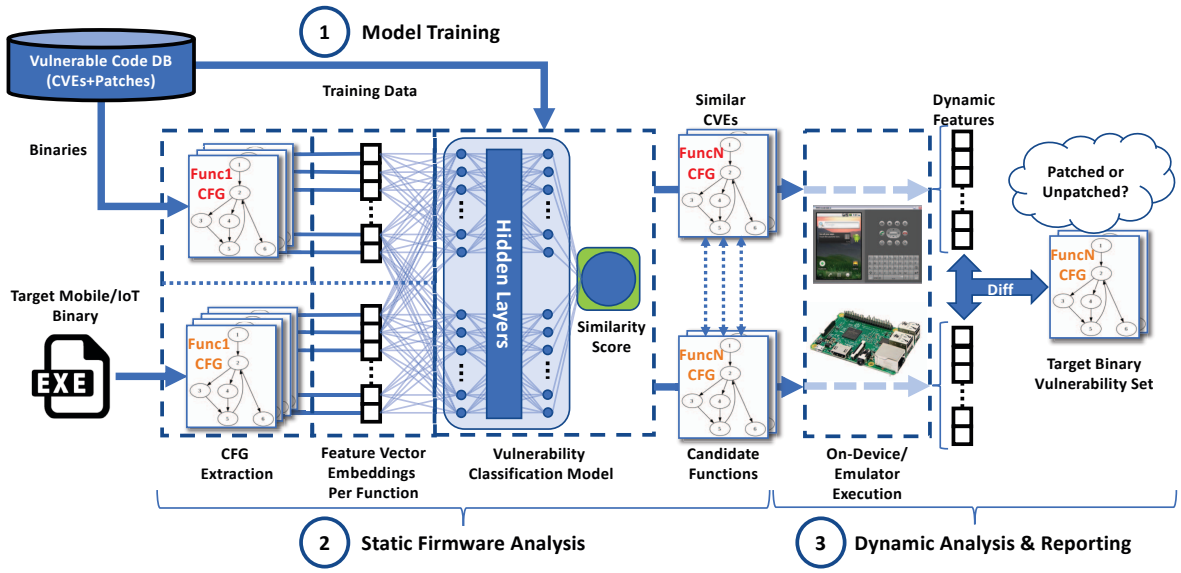


Fig. 1: PATCHECKO vulnerability and patch search workflow.

compilations with different levels of optimization produce different binary programs from identical source code. This way, we can generate one vulnerable function binary for different hardware architectures (e.g., x86 and ARM) and software platforms (e.g., Windows, Linux, and MacOS).

Copious amount of candidate vulnerable functions. To illustrate the scale of the number of candidate vulnerable functions, we analyzed the firmware of Android Things 1.0 and IOS 12.0.1. For Android Things 1.0, we found 379 different libraries that included 440,532 functions, while IOS 12.0.1 contained 198 different libraries with 93,714 functions. Although prior work has shown that deep learning-based methods can be used to identify a set of vulnerable candidate functions with relatively high accuracy [41], these techniques do not present an automated solution to prune/eliminate the resulting false positives. Additionally, the solution in [42] relies on symbol tables that are not available on stripped COTS binaries. As such, there remains a challenge to prune vulnerable candidate functions for stripped COTS binaries.

Differentiating between patched or vulnerable code. Vulnerable functions may not be very distinguishable from their patched versions as a patch may be as little as changing a single line of code. Past work [42] can detect whether or not vulnerable code has been patched. However, this solution relies on access to the source code for both the vulnerable code as well as the patched code. In practice, we often do not have access to the source code of binary functions.

Given these motivating challenges, we now present an overview of the PATCHECKO framework.

B. Approach

An overview of the PATCHECKO framework is presented in Figure 1. Our solution is implemented in three steps: (1) deep learning is used to train the vulnerability detector; (2) the vulnerability detector is used to statically analyze the target mobile/IoT firmware; (3) the identified vulnerable subroutines are run for in-depth dynamic analysis and verification of the existence of a vulnerability. The analyses use the extracted static and dynamic features of vulnerable and patched functions to identify whether the candidate vulnerability function has been patched.

PATCHECKO's objective is to compare the functions within firmware binaries to the set of known CVE vulnerabilities as well as any associated patches. PATCHECKO outputs the vulnerable points (functions) within the target firmware image and the corresponding CVE numbers. To compare two binary functions at runtime, PATCHECKO combines static and dynamic programming language analysis techniques along with deep learning methods from AI and machine learning. PATCHECKO starts with lightweight static analysis to convert each function within a binary to a machine learning feature vector. PATCHECKO then leverages a previously trained deep neural network model to determine if the two functions (one from the firmware binary and the other one from the CVE database) are similar, i.e., coming from the same source code with possibly different compilation flags. If the two functions are detected to be similar, PATCHECKO performs a more in-depth dynamic analysis to ensure the report by the static analysis is not a false positive and indeed indicates a matching function pair.

To perform dynamic analysis, PATCHECKO leverages runtime DLL binary injection and remote debugging solutions to run the CVE vulnerable function binary as well as the target firmware function binary on identical input values (e.g., function arguments and/or global variables) within the corresponding mobile/IoT embedded system platform. PATCHECKO captures the execution traces of the two function binary executions and extracts dynamic features such as number/type of executed instructions, number/type of system calls and library function calls, amount of stack/heap data read/writes, etc. for each execution trace.

Using the extracted features, PATCHECKO calculates a similarity measure between the two functions and determines whether the report by the static analysis is indeed correct. If so, the target function within the firmware is reported to be

vulnerable along with the corresponding CVE number. It is noteworthy that PATCHECKO's analysis is performed without any source code access and hence its deployment does not rely on the cooperation of the firmware vendors.

Since we don't really know whether the reported function is patched, PATCHECKO will first compare the difference based on their static features and restart the whole process based on the patched version of the vulnerable function. PATCHECKO then uses the differential engine to analyze the static/dynamic features as well as the similarity score to decide whether the function has been patched.

III. DESIGN

In this section we present the design of the PATCHECKO framework that explores any given mobile/IoT firmware binary executable and discovers and reports vulnerable points in the binary code/data segments of the firmware without access to its source code. Beyond similarity-based vulnerable code discovery, PATCHECKO can also accurately verify the presence/absence of a security patch for a target firmware binary.

A. Deep Learning-Based Firmware Assessment for Known Vulnerabilities

Comparing with the previous bipartite graph matching [44] and dynamic similarity testing [15], deep learning approaches [41] can achieve significantly better accuracy and efficiency for known vulnerability discovery. This is due to the fact that deep learning approaches can evaluate graphical representations of binaries as a whole and can also automatically learn relationships without manually defined rules. PATCHECKO uses a deep learning approach as a first step to generate a list of vulnerable candidate functions on the order of seconds. However, in order to accommodate our prior assumptions, we first need to build a training dataset that extracts static function features to train a deep learning model.

Feature extractor. In order to extract static function features, PATCHECKO first analyzes functions in assembly format. Marking the correct boundary, scope, and range of each assembly routine is usually the first problem to solve. Furthermore, distinguishing between code and data is equally important. The input for PATCHECKO's neural network model is the function feature vector that is extracted from the disassembled binary code of the target function. To obtain this feature vector, we first identify the function boundaries. Function boundary identification with minimal reliance of instruction set semantics is an independent problem of interest. Previous approaches range from traditional machine learning techniques [4] to neural networks [38] to applying function interface verification [35]. In this work, we assume that these steps are handled by the disassembler using a robust heuristic technique. A disassembler can provide the control flow graph (CFG) of a binary–a common feature used in vulnerability detection.

Figure 2 shows the procedure for PATCHECKO's function feature extraction. PATCHECKO utilizes the CFG with different basic block-level attributes as the features to model the function in our problem. For each function, PATCHECKO can extract function-level, basic block-level and inter-block-level information. Table I shows the completed extracted interesting 48 features from each function for generating a feature vector.

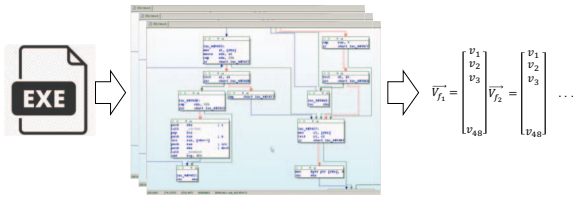


Fig. 2: PATCHECKO's static analysis of mobile/IoT firmware.

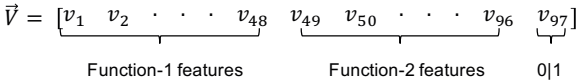


Fig. 3: Sample feature vector for deep learning model.

PATCHECKO keeps the feature extraction rich (48 features), efficient (automated feature extraction) and scalable (multi-architecture support).

Training the deep learning model. For PATCHECKO's deep learning, we adapt a sequential model that is composed of a linear stack of layers. All hyperparameters were determined empirically. Figure 3 shows a sample vector for training the deep learning model. The sample vector is composed of the function vector pairs and a bit indicating whether the two functions are similar. Two similar feature vectors correspond to the two subroutine (function) binaries that come from the same source codes. Figure 4 depicts an actual example process of training the model with a 6-layer network. We first specify the input for each layer. The first layer in our sequential model needs to receive information about its input shape. The model is trained using the extracted function features in our dataset built from 2,108 binaries with different architectures.

B. Pruning Candidate Functions (False Positives) via In-Depth Dynamic Analysis

We use dynamic analysis to further prune the candidates returned by the deep learning stage. This step determines whether the reported pair of matching functions from the previous stage are indeed a match (i.e. either a patched or vulnerable function). This dynamic analysis step executes candidate functions of the two binaries with the same inputs and compares the observed behaviors and several features for similarity.

Two functions may be compiled with different flags. In that case, the instruction execution traces of these functions may differ drastically for the same input. Hence, our analysis will consider the semantic similarity of the execution traces in terms of the ultimate effect on the memory after the two functions finish their execution on the identical input values.

To do so, we extract features from the execution traces (*dynamic features*). Our approach compares the feature vectors of the two traces that result from two function executions on the same input values. If the observed features are similar across different generated inputs, we assume that they are semantically similar.

Figure 5 illustrates the workflow of PATCHECKO's dynamic analysis. There are a few challenges to apply the dynamic analysis for actual execution. At first, one resides in preparing the execution environment. The second one simultaneously

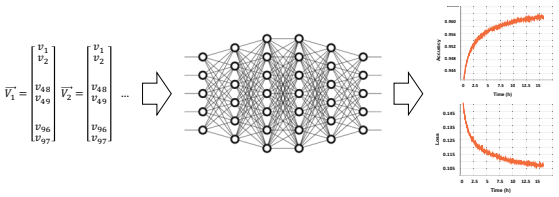


Fig. 4: Training the neural networks for automated firmware vulnerability assessment of mobile/IoT firmware.

TABLE I: Function features used in PATCHECKO.

Feature Name	Feature Description
num_constant	the number of constants value in the function
num_string	the number of strings in the function
num_inst	the number of instruction in the function
size_local	the size of local variables in bytes
fun_flag	various flags associated with a function, e.g., FUNC_NORET, FUNC_FAR.
num_import	the number of import functions
num_ox	the number of code references from this function
num_cx	the number of function calls from this function
size_fun	the size of the function
min_i_b	the minimal number of instruction for basic block
max_i_b	the maximal number of instruction for basic block
avg_i_b	the average number of instruction for basic block
std_i_b	the standard deviation of number of instruction for basic block
min_s_b	the minimal size of basic block
max_s_b	the maximal size of basic block
avg_s_b	the average size of basic block
std_s_b	the standard deviation of size of basic block
num_bb	the number of basic block for each function
num_edge	the number of edge of among basic blocks for each function
cyclomatic_complexity	function cyclomatic_complexity = Edges - Nodes + 2
fcb_normal	normal block type of function basic block
fcb_indjump	block ends with indirect jump
fcb_ret	return block type of function basic block
fcb_cndret	conditional return block type of function basic block
fcb_noret	noreturn block type of function basic block
fcb_enoret	external noreturn block (does not belong to the function)
fcb_extern	external normal block type of function basic block
fcb_error	block passes execution past the function end
min_call_b	the minimal number of call instruction of each basic block
max_call_b	the maximal number of call instruction of each basic block
avg_call_b	the average number of call instruction of each basic block
std_call_b	the standard deviation of call instruction of basic block
sum_call_b	the total number of call instruction of the function
min_arith_b	the minimal number of arithmetic instruction of each basic block
max_arith_b	the maximal number of arithmetic instruction of each basic block
avg_arith_b	the average number of arithmetic instruction of each basic block
std_arith_b	the standard deviation of arithmetic instruction of each basic block
sum_arith_b	the total number of arithmetic instruction of the function
min_arith_fp_b	the minimal number of arithmetic FP instruction of each basic block
max_arith_fp_b	the maximal number of arithmetic FP instruction of each basic block
avg_arith_fp_b	the average number of arithmetic FP instruction of each basic block
std_arith_fp_b	the standard deviation number of arithmetic FP instruction of each basic block
sum_arith_fp_b	the total number of arithmetic FP instruction of the function
min_betweeness_cent	the minimal number of betweeness centrality
max_betweeness_cent	the maximal number of betweeness centrality
avg_betweeness_cent	the average number of betweeness centrality
std_betweeness_cent	the standard deviation number of betweeness centrality
betweeness_cent_zero	how many node the betweeness centrality is zero

monitors the execution of multiple candidate functions. Furthermore, because we are working in a heterogeneous mobile/IoT ecosystem, concretely running binary code to obtain execution traces is not trivial, especially since "valid" values are required for correct function execution. We first discuss the preparation of the inputs that feed into the dynamic analysis engine.

Inputs to the dynamic analysis engine. A key challenge in implementing PATCHECKO's dynamic analysis engine consists of preparing the associated inputs. The dynamic analysis engine takes two inputs: the program binary, **F**, and the *execution environment* of **F**. The program binary contains the target function, `f`. One difficulty consists in isolating the binary execution to the target function. One approach consists of providing concrete and valid input values. This usually requires loading and executing the entire program binary since it is not possible to instruct the operating system to start execution at a particular address. PATCHECKO solves this problem by providing an *execution environment* that contains the required execution state.

PATCHECKO uses fuzzing to generate different inputs for

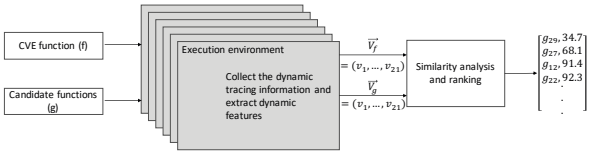


Fig. 5: PATCHECKO's dynamic analysis. Given CVE function f and a set of candidate functions g, collect the dynamic features on given execution environments and compute the similarity between CVE function and candidate functions based on dynamic features vector.

target functions to boost coverage of the associated CFG. For each execution of a target function, PATCHECKO exports a compact representation of a function-level executable, i.e., a compact binary representation of the file that can be executed dynamically using runtime DLL binary injection, as well as the associated inputs that triggered that execution. This allows the dynamic analysis execution engine to efficiently execute the target function. This implies that PATCHECKO will use multiple fixed execution environments associated with different inputs for target functions.

Candidate functions execution validation. Before PATCHECKO begins to instrument the target function execution, PATCHECKO uses multiple fixed execution environments to perform execution on a large number of candidate functions. There are several possible outcomes after we start to run a target function, `f`. For example, the candidate `f` may terminate, the candidate `f` may trigger a system exception, or the candidate `f` may go into an infinite loop. If the candidate `f` triggers a system exception, we will remove the candidate function from a candidate set. After validating candidate functions execution with multiple execution environments, the reserved candidate functions will be instrumented.

Target function instrumentation. The output of the dynamic analysis engine for a function `f` in a fixed execution environment is a feature vector $v(f, env)$ of length N. In order to generate the feature vector, PATCHECKO traces the function execution. For the actual dynamic analysis, a wealth of systems are available such as debuggers, emulators, and virtual machines. However, because of the heterogeneity of mobile/IoT firmware architectures and platforms, PATCHECKO utilizes an instrumentation tool that supports a variety of architectures and platforms accordingly.

PATCHECKO extracts particular features that capture a variety of instruction information (e.g., number of instructions), system level information (e.g., memory accesses), as well as higher level attributes such as function and system calls. Table II shows the initial set of features we initially considered and eventually proved to be useful for establishing function binary similarity. However, this feature list is not comprehensive and can easily be extended.

For each execution, the dynamic engine will generate a set of observations for each feature, e.g., in the above case there will be 21 sets of observations. Once all instructions for a function `f` have been covered, PATCHECKO combines the observations into a single vector, e.g., ($\mathtt{f}_{\mathtt{input\_1}}$). The same process is repeated for different inputs for the same function to produce ($\mathtt{f}_{\mathtt{input\_2}}$),($\mathtt{f}_{\mathtt{input\_3}}$),...,($\mathtt{f}_{\mathtt{input\_N}}$).

TABLE II: Dynamic features used in PATCHECKO.

Index	Feature Name	Feature Description
1	binary_defined_fun_call_num	number of binary-defined function calls during execution
2	min_stack_depth	the minimal stack depth during execution
3	max_stack_depth	the maximal stack depth during execution
4	avg_stack_depth	the average stack depth during execution
5	std_stack_depth	the standard deviation stack depth during execution
6	instruction_num	number of executed instruction
7	unique_instruction_num	number of executed unique instruction
8	call_instruction_num	number of call instruction
9	arithmetic_instruction_num	number of arithmetic instruction
10	branch_instruction_num	number of branch instruction
11	load_instruction_num	number of load instruction
12	store_instruction_num	number of store instruction
13	max_branch_frequency	the maximal number of frequency of the executed same branch instruction
14	max_arith_frequency	the maximal number of frequency of the executed same arithmetic instruction
15	mem_heap_access	number of accessing heap memory space
16	mem_stack_access	number of accessing stack memory space
17	mem_lib_access	number of accessing library memory space
18	mem_anon_access	number of accessing anonymous mapping memory space
19	mem_others_access	number of accessing others part memory space
20	library_call_num	number of library function calls during execution
21	syscall_num	number of system calls during execution

Now that we have the capability of extracting dynamic features of a target function, we next present the algorithm for calculating function similarity for a given pair of functions and their extracted feature sets.

C. Calculating Function Semantic Similarity

For each function pair, (f, g), PATCHECKO computes a semantic similarity measure based on the dynamic feature vector *distance* between the two functions. Distance has been used in data mining contexts with dimensions representing features of the objects. In particular, PATCHECKO uses Minkowski distance [37] as our similarity measures based on each function's feature vector. Different behaviors result in slightly different values of the corresponding coordinates in the feature vectors. We now explore the distance measure in detail.

The Minkowski distance is a generalized form of the Euclidean distance (if p=2) and of the Manhattan distance (if p=1). In our case, we set p=3 for Minkowski distance. The general equation is as follows,

$$d_k(f,g) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}. \quad (1)$$

In Minkowski distance equation, f represents the CVE function and g represents the candidate function in the target firmware. k is the k-th execution environment used. x represents the dynamic feature vector of f and y represents the dynamic feature vector of g. P is set to 3.

We compute the similarity of each pair of (f,g) in multiple execution environments. So we compute their similarity by averaging the similarity distance over the execution environments. We set *K* as the number of execution environments used. We define

$$sim(f,g) = \frac{1}{K} \sum_{i=1}^{K} d_k(f,g). \quad (2)$$

Finally, we feed the dynamic feature vector of each candidate function into the similarity computing equation. We can get a list of ranking (function, similarity distance) pairs (see Figure 5). This is the final component for the identification of known vulnerabilities. We now design the final component that allows us to perform patch presence detection.

D. Patch Detection

We noticed that a patch typically introduces few changes to a vulnerable function. However, these minor changes can still have a significant impact to make the pre- and post-patch functions dissimilar - this intuition is confirmed in Section V. Based on this notion, PATCHECKO uses a *differential engine* to collect both static and dynamic similarity measures in order to determine if a vulnerable function has been patched.

Given a vulnerable function f_v, a patched function f_p, and a target function f_t, the differential engine will first generate three values: the *static features* of f_v, f_p, and f_t, and the *dynamic semantic similarity scores* of sim_v vs. sim_t and sim_p vs. sim_t, as well as the *differential signature* between S_v and S_p. The static features are the same aforementioned 48 different quantified features and the dynamic semantic similarity scores are the aforementioned function similarity metrics. The differential signatures are an additional metric that compares the CFG structures, i.e., the CFG topologies of two functions as well as the semantic information, e.g., function parameters, local variables, and library function calls.

IV. IMPLEMENTATION AND CASE-STUDY

We implemented the PATCHECKO framework on Ubuntu 18.04 in its AMD64 flavor. Our experiments are conducted on a server equipped with one Intel Xeon E51650v4 CPU running at 128 GB memory, 2TB SSD, and 4 NVIDIA 1080 Ti GPU. During both training and evaluation, 4 GPU cards were used. As in the design, PATCHECKO consists of four main components: a feature extractor, a deep learning model, a dynamic analysis engine, and a differential analysis engine for patch detection.

A. Feature Extractor and Deep Learning

The input for the feature extractor is the disassembled binary code of the target function. We assume the availability and the correctness of function boundaries by building on top of IDA Pro [19], a commercial disassembler tool used for extracting binary program features. As such, we implemented the feature extractor as a plugin for IDA Pro. We developed two versions of the plugin: a GUI-version and command line-version (for automation). Since PATCHECKO works on cross-platform binaries, the plugin can support different architectures (x86, amd64 and ARM 32/64 bit) for feature extraction.

We implement the neural network modeling, training and classification based on Keras [8] and TensorFlow [2]. We use TensorBoard [6] to visualize the whole training procedure.

B. Dynamic Analysis Engine

As was mentioned in the design section, the key challenges for dynamic analysis are the preparation of the inputs for the engine as well as the instrumentation of target functions for tracing dynamic information.

Input preparation. As was mentioned in Section III-B, PATCHECKO needs to efficiently prepare the execution environment. To perform dynamic analysis without having to load the entire binary, we utilize DLL injection to execute compact execution binaries that correspond to a single target function. In particular, we use the dynamic loading function (e.g. `dlopen()`) to load the dynamic shared object binary file which returns an opaque "handle" for the loaded object. This handle is employed with other useful functions in the `dlopen` API, such as `dlsym`. Using `dlsym`, we can directly find the exported functions based on the exported function's name. We then execute the targeted function.

Of course, a library binary will contain a large number of different functions, some of which are non-exported functions. As such, we must find a way to export these functions for further analysis. PATCHECKO uses LIEF [36] to export functions into executable binaries. Such a transformation allows PATCHECKO to instrument a candidate function that was found at a given address by using `dlopen` and `dlsym`. Thus, any candidate function can be exported and executed without running the whole binary. This approach has excellent reliability and efficiency, since we can focus on targeted function without having to spawn the entire binary. Furthermore, we use LibFuzzer [22] to fuzz candidate functions and generate different input sets. For the execution environment, we manually choose concrete initial values for different global variables.

Instrumentation. Because we are targeting heterogeneous mobile/IoT ecosystems, we choose to implement the same instrumentation of PATCHECKO on two dynamic instrumentation frameworks: IDA Pro and GDB. Specifically, we implemented a plugin based on GDB and GDBServer for Android and Android Things platforms, and a plugin based on IDA Pro and debugserver for IOS platforms.

C. Case Study

To facilitate the understanding of our implementation details, we will provide an ongoing example to show how we can locate a known CVE vulnerability and how we can ensure whether the vulnerability has been patched or not patched in Android Things firmwares. Android Things is an embedded IoT-specific operating system platform by Google.

Known CVE vulnerability function discovery. We chose one CVE vulnerability, `CVE-2018-9412`, from Android Security Bulletins [34]. This is a DoS vulnerability in the function removeUnsynchronization of the library `libstagefright`. In order to simplify the case study, we generated these binaries directly from the source codes of both the vulnerable and patched `libstagefright` libraries. We compiled both versions using Clang with optimization level O0. Although PATCHECKO never uses the source code for its analysis, Figure 6 shows the source code and assembly code of the patched `CVE-2018-9412` for illustration. We elaborate on the components of this figure in the following subsection.

Generating a training dataset. We compiled 100 Android libraries from their source code using Clang. The compiler is set to emit code in x86, amd64, ARM 32-bit, and ARM 64-bit with optimization levels O0, O1, O2, O3, Oz, Ofast. In total, we obtained 2,108 library binary files[1]. We provide more details in Section V.

Feature extraction. We use our feature extraction plugin to extract the features on top of IDA Pro. Once we get the raw features, PATCHECKO will refine the raw features to generate the feature vector. PATCHECKO extracted all function features from the library `libstagefright.so` and identified a total of 5,646 functions and generated 5,646 function feature vectors.

Vulnerability detection by deep learning. Once the features are extracted, we use the training model for detection. We also use the vulnerable and patched functions as a baseline. Our model identified 252 candidate functions that are based on the vulnerable function's feature vector while generating 971 candidate functions based on the patched function's feature vector. We also compared the feature vectors of the vulnerable and patched functions to check whether they are similar and found them to be dissimilar, i.e., the patched version has significantly different features than the vulnerable version. Looking at the source code in Figure 6, one can intuit that the patched version is significantly different. For instance, the patch removed the `memmove` function and added one more `if` condition for value checking. Similarly, one can observe the difference in the number of basic blocks at the assembly level.

Dynamic analysis engine. Not only are the numbers of vulnerable candidate functions (252) and patched functions (971) from the last step very large, but the candidate functions are also very similar. As such, it would be difficult to locate the target vulnerability function by manual inspection. We therefore use the dynamic analysis engine to generate dynamic information for each function. We first use LibFuzzer to generate the different inputs for the vulnerability function `removeUnsynchronization`. We tested that these inputs worked with both the vulnerable and patched functions. As before, we use the input to test each candidate function and remove any functions that crashed. Using the input-function validation, we obtain 38 candidate functions for the vulnerable function and 327 candidate functions for the patched function. For these candidate functions, PATCHECKO's dynamic

[1]Some compiler optimization levels didn't work for certain instances

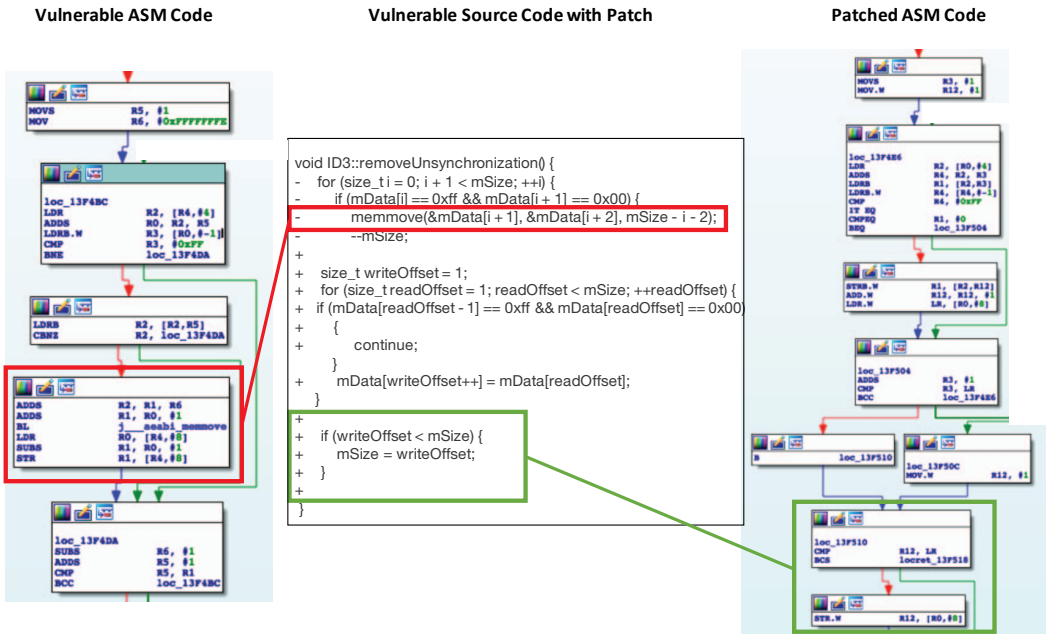


Fig. 6: Vulnerable code with the associated patch of CVE-2018-9412.

analysis engine will generate the dynamic information. For instrumentation in Android Things, we use gdbserver to collect the dynamic features on the Android Things device. Table III shows the part of dynamic feature vector profiling for the vulnerable candidate functions. In the next subsection, we analyze why `candidate_29` is the vulnerable function.

Calculating function similarity. We use the three aforementioned similarity metrics to calculate the function similarity. The top 10 ranking results for the vulnerable function are listed in Table IV and the top 10 ranking results for the patched functions are listed in Table V. For the vulnerable function results, we see that `candidate_29` is the top-ranked candidate, i.e., according to the rule of similarity distance algorithm, if this distance is small, there will be a high degree of similarity. We can also see a significant difference between the top candidate and the second candidate (`candidate_27`). As such, we conclude that `candidate_29` is the vulnerable function.

Diving deeper into the results in Table III, we can observe why the distance between the dynamic features is so small. The two highlighted rows indicate `candidate_29` and the ground truth vulnerable function. Referring back to Table II, we know that *F_13* represents the max frequency for the same branch instruction and *F_14* represents the max frequency for the same arithmetic instruction. We can that `candidate_29` is the only candidate function that has the same frequency numbers as the vulnerable function. It is important to note that this analysis was only enabled by dynamic analysis–static analysis would not have been able to identify these dynamic features.

For the patched case, Table V only shows the results for the top 10 ranking candidate functions due to page limitations. In this case, `candidate_102`, on average, is the top-ranked candidate despite being the incorrect function. However, we can see that `candidate_29` is ranked in a very close second, while there is a significant difference with the third candidate. Intuitively, we can narrow down the candidate functions to the top two and can assume that `candidate_29` is likely to be the associated candidate vulnerable function. However, at this point we cannot tell whether the function is patched.

Differential analysis engine. According the previous steps, we can consider `candidate_29` is the target function. But it is still not clear whether it is patched. We collect static features (e.g. `j___aeabi_memmove`), dynamic semantic similarity scores (`34.7` V.S. `65.6`), and the differential signatures (`j___aeabi_memmove`, `if` condition). Based on these metrics, the differential analysis engine concludes the target function is still vulnerable and not patched.

V. Evaluation

In this section, we evaluate PATCHECKO with respect to its search accuracy and computation efficiency. In particular, we evaluate the accuracy of our deep learning model, the dynamic analysis engine, and the differential analysis engine using a dataset containing ground truth.

A. Data Preparation

In our evaluation, we collected three datasets: 1) Dataset I for training the deep learning model and evaluating the accuracy of the deep learning model; 2) Dataset II for collecting known CVE vulnerabilities and for building our vulnerability database. 3) Dataset III for evaluating the accuracy and performance of the deep learning model, the dynamic analysis engine, and the differential analysis engine for real world mobile/IoT firmware;

Dataset I: This dataset is used for neural network training and baseline comparison. It consists of binaries compiled from source code, providing us with the ground truth. We consider two functions compiled from the same source code function are similar, and dissimilar if they are from different

TABLE III: The dynamic feature vector profiling for part of candidate functions of the vulnerable version of `removeUnsynchronization` in the library `libstagefright.so`. F_1,...,F_21 represents different dynamic features 1 to 21 showed in Table II. In the last row, the vulnerable function is from our vulnerability database.

Candidate	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_10	F_11	F_12	F_13	F_14	F_15	F_16	F_17	F_18	F_19	F_20	F_21
candidate_1	1	2	2	2	0	12	12	0	0	0	4	0	0	0	0	0	0	3	0	0	0
candidate_28	1	2	6	2	1	16	16	0	3	0	1	0	0	1	0	0	0	1	0	0	0
candidate_29	1	2	2	2	0	89	17	0	27	19	19	0	9	9	0	0	0	10	0	1	0
candidate_30	1	2	2	2	0	3	3	1	0	1	0	0	1	0	0	0	0	0	0	0	0
candidate_31	1	2	2	2	0	13	13	0	0	1	1	4	1	0	0	0	0	5	0	0	0
candidate_32	1	2	2	2	0	5	5	0	0	0	1	0	0	0	0	0	0	1	0	0	0
candidate_33	1	2	2	2	0	12	12	0	3	1	2	1	1	1	0	2	0	0	0	0	0
candidate_34	1	2	2	2	0	238	17	44	48	0	0	4	0	40	0	0	0	0	4	0	0
candidate_35	1	2	2	2	0	4	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0
candidate_36	1	2	2	2	0	11	11	0	0	0	4	0	0	0	0	0	0	4	0	0	0
candidate_37	1	2	2	2	0	11	11	1	2	0	2	1	0	1	0	3	0	0	0	0	0
candidate_38	2	2	2	2	0	15	15	0	3	1	5	1	1	1	0	1	2	2	1	0	0
Vulnerable function	1	2	2	2	0	122	21	0	9	18	19	0	9	9	0	0	0	10	0	1	0

TABLE IV: Calculating Function Similarity in PATCHECKO for CVE-2018-9412 in Android Things based on **vulnerable** function for top 10.

Candidate	Sim	Ground truth
candidate_29	**34.7**	**_ZN7android3ID323removeUnsynchronizationEv**
candidate_27	68.1	safe_malloc_mul_2op_p
candidate_12	91.4	ScaleOffset
candidate_22	92.3	_ZNK9mkvparser7Segment11DoneParsingEv
candidate_24	92.3	_ZN9mkvparser15UnserializeUIntEPNS_10IMkvReaderExx
candidate_3	93.3	_ZN9mkvparser6ReadIDEPNS_10IMkvReaderExRl
candidate_7	95.3	_ZN7android8RSFilterD2Ev
candidate_9	95.3	_ZN9mkvparser14UnserializeIntEPNS_10IMkvReaderExxRx
candidate_25	99.2	_ZNK9mkvparser5Block11GetTimeCodeEPKNS_7ClusterE
candidate_28	106.4	_ZN9mkvparser10EBMLHeader4InitEv

TABLE V: Calculating Function Similarity in PATCHECKO for CVE-2018-9412 in Android Things based on **patched** function for top 10.

Candidate	Sim	Ground truth
candidate_102	32.8	CanonicalFourCC
candidate_29	**65.6**	**_ZN7android3ID323removeUnsynchronizationEv**
candidate_52	91.4	_ZN7android11MPEG4Writer13writeLatitudeEi
candidate_76	92.4	_ZN7android11MPEG4Writer14writeLongitudeEi
candidate_85	96.7	_ZN7android21ElementaryStreamQueueC2ENS0_4ModeEj
candidate_93	86.8	__divdf3
candidate_101	106.3	_ZN7android10MediaMuxerC2EiNS0_12OutputFormatE
candidate_40	109.5	ARGBToARGB4444Row_C
candidate_66	113.2	CopyPlane
candidate_111	116.7	_ZN7android10WebmWriter16estimateCuesSizeEi

functions. In particular, we compile 100 Android libraries from their source code (version android-8.1.0_r36) using Clang. We exported 24 different binaries for each Android library by setting the compiler to emit code in x86, AMD64, ARM 32bit, and ARM 64bit ISA with optimization levels O0, O1, O2, O3, Oz, Ofast. However, not every library could be compiled with the six optimization levels, e.g., libbrillo, libbacktrace, libtextclassifier, and libmediaplayerservice. In total, we obtain 2,108 library binary files containing 2,037,772 function feature samples. For this dataset, we compiled all binaries with a debug flag to establish ground truth based on the symbol names. For our problem setting, we strip all binaries before processing them with PATCHECKO.

Dataset II: Since we perform vulnerability assessment, we generated a vulnerability database that includes the static feature vectors and dynamic feature vectors for vulnerable versions and patched versions of functions. The vulnerable function dataset comes from Android Security Bulletins [34]. We collect the vulnerabilities from 07/2016 to 11/2018. In total, there are 2,076 vulnerabilities, including 1,351 high vulnerabilities and 381 critical vulnerabilities.

Dataset III: To evaluate PATCHECKO, we collected different firmware images, which included different versions of Android, Android Things, and IOS. In particular, we select two firmware images from Android Things 1.0 and Google Pixel 2 XL (Android 8.0) as our targets. For vulnerability detection, we considered the vulnerabilities which were patched in 2018 and focus on version 8.0 and 8.1. Finally, we choose 25 different CVE vulnerabilities from our database to evaluate our solution on Android Things and Google Pixel 2 XL.

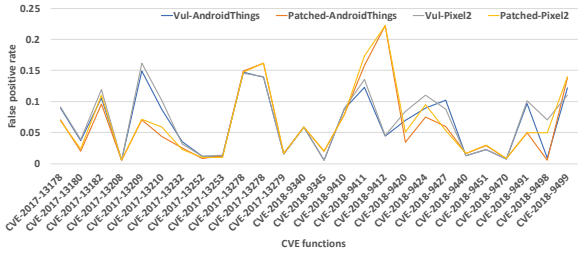
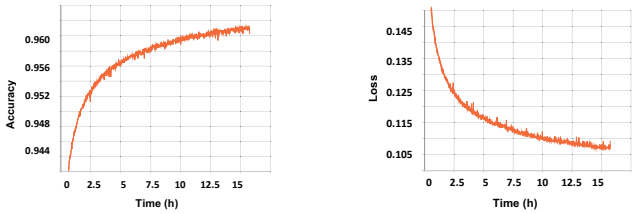


Fig. 7: False positive rate on Android Things and Google Pixel 2 XL with vulnerable and patched versions.



(a) Accuracy for training the deep learning model

(b) Loss for training the deep learning model

Fig. 8: Deep learning training results

B. Training Details

Our deep learning model is first trained using Dataset I. We adapt the sequential model with 6 layers. We first specify the inputs for each layer. The first layer in our sequential model needs to receive information about its input shape. In our case, it is 96. We split Dataset I into three disjoint subsets of functions for training (1,222,663), validation (407,554), and testing (407,555), respectively.

C. Testing Devices

We evaluate PATCHECKO in two different devices: Android Things and Google Pixel 2 XL. For Android Things, we use Android Things 1.0–which includes a 05/2018 security patch as well as a previous security patch. For Google Pixel 2 XL, its system version is Android 8.0 and it includes a 07/2017 security patch as well as a previous security patch.

TABLE VI: The accuracy for deep learning and dynamic execution for Android Things based on vulnerable function. Dp: Deep learning; DA: Dynamic analysis

CVE	Deep Learning Classification						Dynamic Analysis Engine		Time(s)	
	TP	TN	FP	FN	Total	FP(%)	Execution	Ranking	DP	DA
CVE-2018-9451	1	1155	27	0	1183	2.28%	5	1	2.26	187.97
CVE-2018-9340	1	1113	69	0	1183	5.83%	6	1	2.14	197.56
CVE-2017-13232	1	951	35	0	987	3.55%	5	2	3.13	147.97
CVE-2018-9345	1	354	2	0	357	0.56%	1	1	2.72	41.13
CVE-2018-9420	1	107	8	0	116	6.90%	1	1	1.53	39.59
CVE-2017-13210	1	105	10	0	116	8.62%	2	1	1.57	73.18
CVE-2018-9470	1	1421	11	0	1433	0.77%	4	1	6.85	148.37
CVE-2017-13209	0	867	152	1	1020	14.90%	9	N/A	5.25	286.34
CVE-2018-9411	1	894	125	0	1020	12.25%	8	1	5.23	256.58
CVE-2017-13252	1	609	7	0	617	1.13%	7	2	3.35	227.15
CVE-2017-13253	1	609	7	0	617	1.13%	5	2	3.39	167.97
CVE-2018-9499	1	541	75	0	617	12.16%	6	1	2.56	210.35
CVE-2018-9424	1	561	55	0	617	8.91%	7	1	3.02	219.45
CVE-2018-9491	1	421	45	0	467	9.64%	3	1	1.19	108.78
CVE-2017-13278	1	2164	373	0	2538	14.70%	20	2	1.93	602.35
CVE-2018-9410	1	595	57	0	653	8.73%	22	1	2.76	671.46
CVE-2017-13208	1	178	1	0	180	0.56%	1	1	1.23	39.32
CVE-2018-9498	1	13598	130	0	13729	0.95%	7	1	5.90	227.15
CVE-2017-13279	1	723	11	0	735	1.50%	6	1	3.40	224.56
CVE-2018-9440	1	725	9	0	735	1.22%	4	1	2.06	156.52
CVE-2018-9427	1	1060	120	0	1181	10.16%	9	1	4.61	296.31
CVE-2017-13178	1	540	53	0	594	8.92%	15	1	2.01	473.89
CVE-2017-13180	1	571	22	0	594	3.70%	5	2	1.23	157.97
CVE-2018-9412	1	5393	252	0	5646	4.46%	38	1	3.54	1124.53
CVE-2017-13182	1	5050	595	0	5646	10.54%	72	3	3.16	2128.16
Average						6.16%			3.04	336.5844

TABLE VII: The accuracy for deep learning and dynamic execution for Android Things based on patched function. Dp: Deep learning; DA: Dynamic analysis

CVE	Deep Learning Classification						Dynamic Analysis Engine		Time(s)	
	TP	TN	FP	FN	Total	FP(%)	Execution	Ranking	DP	DA
CVE-2018-9451	1	1148	34	0	1183	2.87%	8	2	2.29	246.25
CVE-2018-9340	1	1113	69	0	1183	5.83%	6	1	2.07	197.56
CVE-2017-13232	1	961	25	0	987	2.53%	5	1	3.20	177.97
CVE-2018-9345	1	349	7	0	357	1.96%	4	3	1.66	148.37
CVE-2018-9420	1	111	4	0	116	3.45%	1	1	1.50	59.59
CVE-2017-13210	1	110	5	0	116	4.31%	2	1	1.63	91.19
CVE-2018-9470	1	1420	12	0	1433	0.84%	4	1	5.93	160.46
CVE-2017-13209	1	947	72	0	1020	7.06%	7	1	4.07	207.15
CVE-2018-9411	1	858	161	0	1020	15.78%	10	2	4.24	301.23
CVE-2017-13252	1	611	5	0	617	0.81%	6	1	2.33	230.56
CVE-2017-13253	1	608	8	0	617	1.30%	5	2	2.67	165.51
CVE-2018-9499	1	531	85	0	617	13.78%	9	3	2.57	287.65
CVE-2018-9424	1	570	46	0	617	7.46%	5	1	2.01	156.32
CVE-2018-9491	1	443	23	0	467	4.93%	1	1	2.20	45.93
CVE-2017-13278	1	2159	378	0	2538	14.89%	19	1	1.90	587.86
CVE-2018-9410	1	601	51	0	653	7.81%	21	1	2.83	651.45
CVE-2017-13208	1	178	1	0	180	0.56%	1	1	1.08	35.53
CVE-2018-9498	1	13647	81	0	13729	0.59%	6	1	4.89	243.3
CVE-2017-13279	1	722	12	0	735	1.63%	6	1	3.48	236.78
CVE-2018-9440	1	722	12	0	735	1.63%	5	2	4.84	175.52
CVE-2018-9427	1	1110	70	0	1181	5.93%	2	2	4.74	99.18
CVE-2017-13178	1	551	42	0	594	7.07%	13	1	2.86	390.89
CVE-2017-13180	1	581	12	0	594	2.02%	2	1	2.17	71.48
CVE-2018-9412	1	4391	971	0	5646	17.20%	327	2	3.52	8676.91
CVE-2017-13182	1	5103	542	0	5646	9.60%	42	1	3.15	1249.96
Average						5.67%			2.95	595.784

TABLE VIII: The final patch detection results for PATCHECKO in Android Things

CVE	PATCHECKO Result Patched (?)	Ground Truth Patched (?)
CVE-2018-9451	0	0
CVE-2018-9340	0	0
CVE-2017-13232	✓	✓
CVE-2018-9345	0	0
CVE-2018-9420	0	0
CVE-2017-13210	✓	✓
CVE-2018-9470	✓	0
CVE-2017-13209	✓	✓
CVE-2018-9411	0	0
CVE-2017-13252	✓	✓
CVE-2017-13253	✓	✓
CVE-2018-9499	0	0
CVE-2018-9424	0	0
CVE-2018-9491	0	0
CVE-2017-13278	✓	✓
CVE-2018-9410	0	0
CVE-2017-13208	✓	✓
CVE-2018-9498	0	0
CVE-2017-13279	✓	✓
CVE-2018-9440	0	0
CVE-2018-9427	0	0
CVE-2017-13178	0	0
CVE-2017-13180	✓	✓
CVE-2018-9412	0	0
CVE-2017-13182	✓	✓

D. Accuracy

In this section, we evaluate the accuracy of PATCHECKO's deep learning model, dynamic analysis engine, as well as its patch detection.

Deep learning model. Figure 8 shows the accuracy and loss when we train our deep learning model for ˜15 hours. The accuracy can reach 96%.

Since [42] needs the previous similarity checking solutions to locate the target function when the symbol table is not available, the target function may be missed if the vulnerable version and patched version are not similar. To validate this notion, we use our deep learning model to check the similarity between the vulnerable and patched version of the same function for 25 CVEs. We found that there is a possibility that vulnerable and patch versions are not similar based on the deep learning model. For example, if CVE-2018-9345 had been patched, the solution in [42] will miss the target function based on vulnerable function features and, thus, may use the wrong function to detect whether it is patched.

We use the training model to detect 25 CVEs in Android Things and Google Pixel 2 XL. The average detection accuracy is more than 93%. Figure 7 shows the false positive rate when we test vulnerable and patched versions in the two devices' firmware images. It is interesting that the false positive rate of the vulnerable and patched versions of CVE-2017-13209 and CVE-2018-9412 on the two devices are obviously different, which is reflective of their dissimilar result. Furthermore, we notice that because CVE-2017-13209 has been patched, the false positive rate of the patched versions is lower than vulnerable versions. Similarly, CVE-2018-9412 has not been patched and Figure 7 shows the false positive rate of patched version is higher than the vulnerable version. However, Table VI shows the vulnerable version function gets 0 true positives and 1 false negative in Android Things for CVE-2017-13209. This is due to the fact that CVE-2017-13209 has been patched in Android Things. Therefore, when PATCHECKO uses the vulnerable function, the deep learning model may miss the correct target function. Intuitively, it makes sense that a known vulnerability discovery may miss a patched function as a vulnerability.

Dynamic analysis engine. The goal of the dynamic analysis engine is to prune the set of candidate functions. In Table VI and Table VII, the results for the dynamic analysis engine includes only the **Execution** and **Ranking** metrics. Because we want to reduce the number of candidate functions that we need to perform dynamic feature profiling. We use the concrete input of vulnerable functions to validate the candidate functions. As long as the candidate functions can survive the input validation, PATCHECKO will do dynamic feature profiling for the final candidate function. For example, after deep learning, CVE-2018-9412 still has 252 candidate functions. For dynamic analysis, PATCHECKO arranges different inputs of vulnerable functions to validate the candidate functions. After validation, only 38 candidate functions remain that require dynamic feature profiling–which is a much more reasonable number. Finally, PATCHECKO calculates the function similarity score. Table VI and Table VII show that PATCHECKO can rank the target function in the top 3 candidates 100% of the time. The target function is only missed for CVE-2017-13209 since the deep learning model already misses the target function.

Patch detection. According to the differential signature, semantic static features, and the results from Table VI and Table VII, PATCHECKO generates the final results in Table VIII. There is only one missed classification for the patched version of CVE-2018-9470. The reason that this classification was missed was due to the fact that the only difference between vulnerable and patched version is one integer–which is a very minute and difficult to detect.

Limitations. If the difference is very minute between a vulnerable function and a patched function, our similarity measures may not catch the difference, e.g., CVE-2018-9470. The missed classification is due to the fact the static feature and dynamic features do not represent the difference between the vulnerable and patched versions of the code. A solution would be to add more fine-grained features from known vulnerability exploits. However, assuming the associated exploits are available, there is a trade-off in generalizability.

E. Processing Time

Table VI and Table VII respectively list the processing times for the deep learning detection and dynamic analysis for vulnerable and patched functions. The deep learning detection phase takes around 3 seconds on average. The dynamic analysis' execution time varies depending on the number of candidate functions to test and the number of execution environments (program states) to replicate. For example, CVE-2017-13208 takes much less time than CVE-2017-13182 due to the large difference (72) in candidate functions. For the dynamic analysis, PATCHECKO bootstraps the execution environments that correspond to the candidate functions. These environments are run in parallel. PATCHECKO currently parallelizes the execution environment testing for all candidate functions. Future works will focus on parallelizing the candidate function execution in each environment to further reduce the dynamic analysis processing time.

VI. Related Work

We briefly survey the related work. We focus on approaches that use code similarity for known vulnerabilities without access to source code. Other approaches for finding unknown vulnerabilities [3], [5], [39], [7] and source code based approaches [24], [23], [21], [20], [28] will not be discussed in this section. We divide the related work to programming languages and machine learning based solutions.

Programming language-based solutions. The problem of testing whether two pieces of syntactically-different code are semantically identical has received much attention by previous researchers. A lot of traditional approaches are based on a matching algorithm for the CFGs of functions. Bindiff [44] is based on the syntax of code for node matching. At a high-level, BinDiff starts by recovering the control flow graphs of the two binaries and then attempts to use a heuristic to normalize and match the vertices from the graphs. For [33], each vertex of a CFG is represented with an expression tree. Similarity among vertices is computed by using the edit distance between the corresponding expression trees. Another approach that focuses on cross-platform binary similarity [16] proposes a graph-based methodology. It used a matching algorithm on the CFGs of functions. The idea is to transform the binary code into an intermediate representation. For such a representation, the semantics of each CFG vertex are computed by using a sampling of the code executions using random inputs. On the theoretical side, [17], [41] extract feature representations from the control flow graphs and encodes them into graph embeddings to speed up the matching process. Other recent work [31], [25] leverage similar static analysis techniques.

In comparison to static analyses, dynamic binary analysis is another approach to detect function similarity. Binhunt [18] implemented symbolic execution and theorem proving as a dynamic binary differentiate to test basic blocks for semantic differences. However, this method only focused on basic blocks. There is a possibility that two functions are functionally equivalent but they may have different basic blocks due to compiler optimizations. Egele et. al [15] proposed a dynamic equivalence testing primitive that achieves complete coverage by overriding the intended program logic. It collects the side effects of functions during execution under a controlled randomized environment. Two functions can be similar if their side effects are similar. However, it requires executing each function with many different inputs–which is time-consuming.

Machine learning-based solutions. Deep learning-based graph embedding approaches have also been used to do binary similarity checking. The current state-of-the-art [41] looks for the same affected functions in the complete collection of functions in a target binary. Similarly, [29] leverages machine code to compute the best parameters for their neural network model. [43] and [14] both use Natural Language Processing (NLP) to replace the manually selected features. However, when the search space is huge, it still leaves a large set of candidate functions. PATCHECKO can integrate dynamic analysis to prune the candidate functions and reduce false positives.

Zhang et. al [42] proposed a unique position that leverages the source-level information to answer a more specific question: whether a specific affected function is patched in the target binary. However, it needs the source code support as well as the aforementioned similarity checking solutions to help it to locate the target function. PATCHECKO uses deep learning and dynamic binary analysis to locate the target function and perform accurate patch detection.

VII. Conclusion

We presented PATCHECKO, a vulnerability assessment framework that leverages deep learning and hybrid static-dynamic binary analysis to perform cross-platform binary code similarity analysis to identify known vulnerabilities without source code access with high accuracy. PATCHECKO then uses a differential engine to distinguish between vulnerable functions and patched functions. We evaluated PATCHECKO on 25 existing CVE vulnerability functions for Google Pixel 2 smartphone and Android Things IoT firmware images on heterogeneous architectures - we compiled the firmware images for multiple architectures and platforms with different compiler optimization levels. Our deep learning model identifies vulnerabilities with an accuracy of over 93%.

We also demonstrated how dynamic analysis of the vulnerability functions in a controlled environment could be used to significantly reduce the number of candidate functions (i.e., eliminate false positives). PATCHECKO identifies the correct matches (candidate functions) among the top 3 ranked outcomes 100% of the time. Furthermore, PATCHECKO's differential engine distinguishes between functions that are still vulnerable and those that are patched with an accuracy of 96%.

Acknowledgments

We appreciate the Office of Naval Research (ONR) and the National Science Foundation (NSF) for their support of our project.

REFERENCES

[1] "Why are android devices slow to patch?; available at https://duo.com/blog/why-are-android-devices-slow-to-patch," 2018.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," 2011.

[4] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code." USENIX, 2014.

[5] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 725–741.

[6] W. Chargin, "Tensorboard; available at https://www.tensorflow.org/guide/summaries_and_tensorboard," 2017.

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

[8] F. Chollet *et al.*, "Keras: Deep learning library for theano and tensorflow," *URL: https://keras. io/k*, vol. 7, no. 8, 2015.

[9] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proceedings of the 26th USENIX Conference on Security Symposium, Security*, vol. 17, 2017.

[10] C. Cimpanu, "NASA hacked because of unauthorized Raspberry Pi connected to its network," https://www.zdnet.com, 2019, [Online; accessed 12-December-2019].

[11] F. T. COMMISSION, "Mobile security updates: Understanding the issues; available at https://www.ftc.gov/system/files/documents/reports/mobile-security-updates-understanding-issues/mobile_security_updates_understanding_the_issues_publication_final.pdf."

[12] R. Cox, "Our software dependency problem; available at https://research.swtch.com/deps," 2019.

[13] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," 2013.

[14] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization*. IEEE, 2019, p. 0.

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

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

[17] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 480–491.

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

[19] Hex-Rays, "Ida pro disassembler; available at https://www.hex-rays.com/products/ida/."

[20] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 1909–1915.

[21] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, 2016, pp. 1606–1612.

[22] L. C. Infrastructure, "libfuzzer: a library for coverage-guided fuzz testing," 2017.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[24] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[25] 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*. IEEE Press, 2013, pp. 329–338.

[26] D. Labs, "Thirty percent of android devices susceptible to 24 critical vulnerabilities; available at shorturl.at/dhj17."

[27] S. R. Labs, "The android ecosystem contains a hidden patch gap; available at https://srlabs.de/bites/android_patch_gap/."

[28] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

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

[30] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system," *IEEE Security & Privacy*, vol. 4, no. 6, 2006.

[31] B. H. Ng and A. Prakash, "Expose: Discovering potential binary code re-use," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 492–501.

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

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

[34] A. O. S. Project, "Android security bulletins; available at https://source.android.com/security/bulletin."

[35] R. Qiao and R. Sekar, "Effective function recovery for cots binaries using interface verification," Technical report, Secure Systems Lab, Stony Brook University, Tech. Rep., 2016.

[36] Quarkslab, "Lief: Library to instrument executable formats; available at https://lief.quarkslab.com/," 2017.

[37] G. Rossi and M. Testa, "Euclidean versus minkowski short distance," *Physical Review D*, vol. 98, no. 5, p. 054028, 2018.

[38] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks." in *USENIX Security Symposium*, 2015, pp. 611–626.

[39] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[40] TechNavio. (2014) Global industrial control systems (ics) security market 2014-2018. shorturl.at/jprIX.

[41] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.

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

[43] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.

[44] Zynamics, "Bindiff; available at https://www.zynamics.com/software.html," 2011.