Automatic Vulnerability Detection in Embedded Device Firmware and Binary Code: Survey and Layered Taxonomies

ABDULLAH QASEM*, PARIA SHIRANI*, MOURAD DEBBABI, and LINGYU WANG, Concordia University
BERNARD LEBEL, Thales Canada Inc.
BASILE L. AGBA, Institut de recherche d'Hydro-Québec

In the era of the internet of things (IoT), software-enabled inter-connected devices are of paramount importance. The embedded systems are very frequently used in both security and privacy-sensitive applications. However, these software (a.k.a. firmware) very often suffer from a wide range of security vulnerabilities, mainly due to their outdated systems or reusing existing vulnerable libraries, which is evident by the surprising rise in the number of attacks against embedded systems. Therefore, to protect those embedded systems, detecting the presence of vulnerabilities in the large pool of embedded devices and their firmware plays a vital role. To this end, there exist several approaches to identify and trigger potential vulnerabilities within deployed embedded systems firmware. In this survey, we provide a comprehensive review of the state-of-the-art proposals, which detect vulnerabilities in embedded systems and firmware images by employing various analysis techniques, including static analysis, dynamic analysis, symbolic execution, or hybrid approaches. Furthermore, we perform both quantitative and qualitative comparisons between the surveyed approaches. Moreover, we devise taxonomies based on the features used in the literature, a comprehensive summary of different proposed approaches, and the applications of those approaches. Finally, we identify the unresolved challenges and discuss possible future directions in this field of research.

Additional Key Words and Phrases: Binary analysis, embedded system security, firmware analysis, vulnerability detection

ACM Reference format:
Abdullah Qasem, Paria Shirani, Mourad Debbabi, Lingyu Wang, Bernard Lebel, and Basile L. Agba. 2018. Automatic Vulnerability Detection in Embedded Device Firmware and Binary Code: Survey and Layered Taxonomies. 1, 1, Article 1 (July 2018), 40 pages.
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The information technology is growing rapidly, and the underlying software is playing a paramount role in various domains and critical infrastructures, such as the energy sector, chemical sector, nuclear sector, and transportation systems. Due to this wide adoption, the emerging threats from the software vulnerabilities is becoming one of the most concerning security issues for those critical infrastructures. This security concern is evidenced by the number of recent attacks on embedded devices. For instance, the Mirai botnet compromised millions of IoT devices and orchestrated

*Both authors contributed equally to this research.

them to initiate a distributed denial of service (DDoS) attack and to target many domain name system (DNS) servers, which resulted in denial of service for hundreds of thousands websites across the globe [7]. The Reaper malware, which is considered to be an extended version of the Mirai malware, was launched in 2016 and targeted IoT devices with dedicated vulnerabilities instead of focusing on only the credentials [73]. In 2014, Black Energy [119] APT caused a blackout that occurred by taking control of operating stations. Similar threats have been demonstrated in other countries, for instance, taking control over 50 power generators by intruders, may potentially affect the power supply to 93 million US residents [149]. These real-world attacks demonstrate the severe consequences on IoT devices and embedded systems in critical infrastructures.

Software vulnerabilities are identified as one of the biggest reasons for these attacks, and new vulnerabilities are frequently discovered. For instance, the total number of reported vulnerabilities in 2017 has become more than double since 2016, amounting to an increase of 120% [58]. Such vulnerabilities become more dangerous when they appear in popular libraries that are incorporated into various software projects, embedded systems and firmware images. Recently, several publications highlighted the necessity of the firmware image evaluation [57, 63, 137, 140]. Moreover, Cui et *al.* [38] report that many of the firmware updates contain well-known vulnerabilities within the third-party libraries for years. They additionally demonstrate that 80.4% of the vendors release firmware issued with known vulnerabilities. When considering embedded systems, the destructive impacts can be severe as they control critical systems and hence, attacking such devices could result in massive breakdowns of public systems, and severe security and safety consequences nation-wide or even world-wide. For instance, a Foscam IP camera was reported to have 18 zero-day vulnerabilities, such as insecure default credentials, command injection, and stack-based buffer overflow [59].

Frequent discovery of software vulnerabilities can be mainly due to any of the following reasons. First, many software designers use outdated system architectures [83], and hence, their systems are susceptible to more attacks. Furthermore, the applications might have been developed by reusing vulnerable libraries. Such reused libraries are rarely analyzed for vulnerability detection, updated or patched. Second, the internet connectivity, and integration and platform compatibility requirements of embedded systems makes them more likely to be exposed to attacks and misuse. Finally, traditional security tools and existing solutions, such as anti-viruses or firewalls, cannot be employed, since these devices are resource constrained in terms of computing power and available memory. Consequently, adversaries exploit these limitations and build dedicated malware targeting underlying embedded systems and IoT devices.

Software vulnerability detection can be performed on both source code and binary code. The latter approaches (e.g., [69, 84, 87, 91, 157]) rely on the source code for vulnerability identification. However, these approaches are not always practical, since most commercial software products are not open source. As a result, binary code analysis becomes an absolute necessity. At the same time, manual binary analysis is a daunting, error-prone, and challenging task, especially for a large corpus of embedded device firmware images. Therefore, automatic and scalable vulnerability detection becomes crucial; specifically, scanning a large number of firmware images and binaries for well-known or zero-day vulnerabilities as well as generating a security evaluation report in a reasonable time is highly desirable.

Binary analysis on firmware images of embedded systems is even more challenging than normal binary analysis. Since in addition to the challenges related to normal binary analysis, there exists specific challenges to analysis of embedded system firmware images. First, embedded systems are tailored to specific hardware, therefore, in the absence of standards and the presence of various CPU architectures, firmware analysis becomes more challenging. Second, embedded systems are customized for specific tasks and are resource constrained for minimizing the cost and power usage. Consequently, embedded systems have specific efficiency and scalabiltiy issues at runtime, since parallel techniques using multi-processing or virtualization that can accelerate testing campaigns are impractical in

embedded systems compared to desktop environment. Third, there is a limitation in performing fault detection on embedded systems. Due to the limited I/O capabilities, computing power, and cost, the memory corruption attacks are less observable in embedded systems. Only some specific attacks (e.g., format string and stack-based overflow) which cause memory corruption can be more easily detected, since the device crashes. Finally, firmware reverse engineering which involves firmware acquisition, unpacking and binary identification is a specific challenge for embedded systems.

Even though there are a few efforts to examine some state-of-the-art approaches on vulnerability detection in binary code, there is no previous survey focusing on the literature for automatically inspecting vulnerabilities in embedded devices firmware and binary files. Brooks [21] investigates the similarities and dissimilarities of two winner systems proposed for vulnerability detection and exploit generation in the DARPA Cyber Grand Challenge (CGC) [43], and provides datasets and a standard platform for cyber reasoning systems (CRSs) evaluations. Ji et *al.* [86] study existing cyber reasoning systems for vulnerability detection, exploitation and patching. Another work [88] surveys software vulnerability analysis, where the main focus is at the source-code level approaches that employ machine learning techniques. In summary, no prior survey focuses on the current vulnerability detection mechanisms for firmware and binaries in the embedded devices in order to identify their strengths and weaknesses, and finally pose important open research questions on this domain.

In this survey, we accomplish a comprehensive study of binary analysis approaches proposed for automatic inspection of vulnerabilities in embedded systems firmware images and binary code. We systematically compare 71 studies focusing on binary and embedded system firmware analysis and examined about 200 papers in this area, including relevant publications in the last proceedings of major conferences in both cybersecurity and software engineering from Jan 2004 to Jan 2020. The former includes ACM Conference on Computer and Communications Security (CCS), IEEE Symposium on Security and Privacy (S&P), Network and Distributed System Security Symposium (NDSS), USENIX Security Symposium (USEC), Symposium on Research in Attacks, Intrusions and Defenses (RAID), ACM Asia Conference on Computer and Communications Security (ASIACCS) and Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). And the latter includes ACM International Symposium on the Foundations of Software Engineering (FSE), IEEE/ACM International Conference on Automated Software Engineering (ASE), International Conference on Software Engineering (ICSE), and IEEE Transactions on Software Engineering (TSE).

We first study both generic and embedded-specific binary analysis challenges and techniques for automated vulnerability detection (in Section 1). We then study and categorize the existing features that are being used in binary analysis (in Section 4.1), and further categorize the works based on their applications (in Section 2.3). Besides, we examine several existing works in order to gain more insights into the proposed techniques, their strengths and limitations. We devise a taxonomy based on three types of analysis employed in the existing approaches, including dynamic, symbolic execution and static (in Sections 3, 3.2.3 and 4, respectively). Furthermore, we compare them based on different criteria, i.e., methodologies, implementations, and evaluations (in Sections 3.2.4 and 4.2). We observe that most existing works perform the evaluation for specific applications and experimental setup, and hence, they cannot be directly compared. Our observation suggests the need for a common platform (e.g., DARPA CGC) and specific dataset for easier result comparison and for contrasting other capabilities, such as scalability. Additionally, we note that hybrid approaches, which integrate analysis techniques (e.g., static and dynamic), could be more suitable for vulnerability detection. The main contributions of this paper are:

- We perform a comprehensive study of dynamic analysis, symbolic execution, and static vulnerability detection approaches in firmware images and embedded devices.
- We propose taxonomies based on both features and employed approaches.

- We accomplish quantitative and qualitative comparisons between the reviewed approaches.
- We carry out a security gap analysis for the reviewed approaches along with providing recommendations to amend some of the identified gaps.

The rest of this survey is structured as follows: Section 1 provides background on binary analysis. Sections 3, and 4 survey existing dynamic, symbolic executing and static approaches, respectively. Section 5 discusses the lessons learned and future challenges. Section 6 concludes the paper.

2 PRELIMINARIES

In this section, we summarize the main challenges of binary analysis on firmware images for embedded devices. Moreover, we briefly explain major analysis approaches in this domain and relate them to the identified challenges. Then, we propose a taxonomy for application domains, and finally briefly provide a classification for embedded systems.

2.1 Challenges

In the following, we outline the general challenges faced by the binary analysis process and discuss the implication on the analysis of firmware images in embedded devices.

- **C1** - *Information loss*: During the compilation process, some information that is available in the source code, ranging from syntax features (e.g., variable names, and comments), to characteristics of the buffers, and data structures sizes will be lost. Therefore, analyzing the binary code would become more challenging and complicated compared to the source code analysis. Additionally, in the case of stripped binaries where the debugging information (e.g., identifier names) is missing, binary analysis task becomes more challenging.
- **C2** - *Compiler effects*: With the advent of modern compilers and run-time libraries, binary code analysis is becoming a very challenging task. Most compilers apply performance or memory optimizations, which result in significant variation in the binary representations for different configurations; for instance, registers, calling conventions, control flow graphs, and also arithmetic operations might be different. These differences are more significant if another compiler or compilation settings are used, or even if the source code has been slightly modified.
- **C3** - *Binary disassembling*: Binary disassembling is still a challenging task mainly due to the following reasons:
  - *Entry point and function boundary discovery*: The disassembler usually uses the *symbol table* to identify function boundaries and to construct the control flow graphs. However, when the *symbol table* is inaccurate or it is not available, finding function boundaries becomes challenging [163]. Additionally, in some cases such as *binary-blob* firmware [140], the entry point and the base address are not known. Moreover, some functions may have multiple entry points [14], which need to be identified.
  - *Code discovery*: To align instructions and improve cache efficiency, compilers may insert padding bytes between or within the functions. Consequently, it may be hard for the disassembler to distinguish between padded bytes and code bytes, since padded bytes are usually converted into valid instructions [14]. Moreover, some functions may not be continuous and have some gaps including data, jump tables, or instructions from other functions [14], which affect the accuracy of the binary analysis approaches. Additionally, control flow graphs extraction might not be performed accurately due to failing to find all the code, identifying non-return functions, and handling indirect jumps which rely on the computed values.
  - *Code transformation*: The authors of legitimate/benign programs may protect their programs for different reasons, such as intellectual property infringement or preventing their programs from being repackaged and redistributed as malware [1, 155]. Similarly, malware authors apply some techniques on their malicious code to evade analysis

and make them more cumbersome. Obfuscation [35, 100, 162], encryption and packing [94, 146] techniques are used for these purposes to make the binary analysis more challenging and difficult.

- **C4 -** *Function inlining:* A small function might be inlined into its caller function for optimization purposes. The lack of distinction between an inline function and the other parts of the function makes the function inline identification task very challenging. This task becomes more challenging when the assembly instructions of an inline function are discontinuous as the result of instruction alignment and pipelining.
- **C5 -** *Hardware architecture*: Software programs can be cross-compiled or deployed on different CPU architectures, where instruction sets, calling conventions, register sets, function offsets and memory access strategies vary from one architecture to another [129]. Therefore, analyzing binaries compiled for different CPU architectures from the same source code is more challenging.
- **C6 -** *Accuracy*: Achieving high accuracy for any vulnerability detection technique is a critical and non-trivial task. For instance, obtaining low false positive rates is usually challenging when analyzing the code statically [139].
- **C7 -** *Results verification*: For several techniques, the obtained vulnerability detection results cannot be verified due to the limited access to the information of the identified vulnerabilities (e.g., how to trigger). Therefore, these techniques involve manual efforts to verify the results and hence, can be error-prone and inefficient.
- **C8 -** *Efficiency*: Many existing approaches are computationally expensive, which demonstrate the need of efficient vulnerability identification techniques for any binary code.
- **C9 -** *Scalability*: Number of deployed software is increasing exponentially, due to the dramatic growth of desktop applications, IoT devices, embedded systems, and inter-connectivity between them. Vulnerability detection approaches need to deal with a large number of binaries and firmware images. Thus, large scale vulnerability analysis is an absolute requirement.
- **C10 -** *Test case generation*: Some approaches require an initial seed input compatible with the target application to start with the analysis. Test cases are easy to generate when the targeted application provides its required input file format. However, when no configuration input is provided, test case generation becomes a challenging task; since each program needs a particular test case to be prepared in advance, which requires expertise and additional effort.

Challenges Specific to Embedded Systems. Along with the issues detailed for C5 and C9, embedded systems have additional aspects of those challenges as well as two more specific limitations (C11 and C12) as outlined below.

- **C5 -** *Hardware architecture*: Specifically, embedded systems are customized for specific tasks and therefore their firmware are tailored to run on their specific hardware. Due to lack of standards, the wide diversity of architectures further challenges firmware analysis. Thus, a solution for a specific CPU architecture cannot easily be generalized.
- **C9 -** *Scalability*: Embedded systems encounter specific scalability issues compared to normal binaries. More specifically, testing embedded firmware at runtime, either on their target devices or in an emulated environment is very slow [114]. While in desktop environment, parallel techniques using multi-processing or virtualization can accelerate testing campaigns, this would be impractical in embedded systems; since they require access to a batch of similar physical devices which is impractical due to financial cost or limited resources (e.g., power supply and space). Moreover, testing an embedded system requires frequent restarts to ensure a clear state for each new test case.
- **C11 -** *Fault Detection*: While some dynamic approaches (e.g., fuzzing explained in Section 2.2.2) are used on traditional computing environments (e.g., desktops and servers), they have limitations on embedded systems [114] due to missing fault detection implementation on firmware components. In traditional computing environments, the OS implements protection levels (e.g., stack Canaries, fault segmentation, ASLR, heap hardening, and sanitizers) to

prevent memory corruption. Such protections are usually not implemented in embedded systems due to the limited I/O capabilities, computing power, and cost. This makes memory corruption attacks less observable and riskier, as out of bound memory corruption may end-up writing in memory-mapped peripheral registers (e.g. flash erase = 1) or worst if it is embedded in an operational technology (OT), e.g., trigging the brake actuator of a vehicle. Silent memory corruption in an embedded system is a rule, not an exception compared to traditional desktops [114]. The only such attacks observable in the embedded world are those related to format string and stack-based buffer overflow vulnerabilities, since they cause memory corruption and consequently usually the device crashes.

- **C12-** *Firmware reverse engineering:* Firmware reverse engineering can be a time consuming and challenging task and requires domain expertise. The whole process involves the following steps:
  - *Firmware acquisition*: Embedded system vendors tend to avoid publishing their firmware in order to protect their IP or limit the access to it. Therefore, it might be necessary to directly extract or dump it from a device chip memory in different ways, such as an EEPROM programmer, bus monitoring during code upload and schematic extraction [147]. However, hardware locks and component interference might make this task challenging. This can be resolved by physically modifying the original hardware, or manipulating the circuit boards using probes.
  - *Firmware unpacking and extraction*: Some vendors pack their firmware using proprietary packers and file formats, or use private key encryption. In practice, different unpacking tools, such as Binwalk [77], BAT [79], and FRAK [38] can be utilized to extract the firmware. However, performing such tasks has limited success rate and thus not all embedded device firmware can be analyzed (e.g., Costen et al. [28] successfully unpacked 8,617 firmware out of 23,035 collected firmware images).
  - *Firmware and binary identification*: Once the firmware is extracted, filtering is required for obtaining all relevant information. This can include binary files, configuration files, embedded files and the firmware itself. To this end, file signature matching is performed using different tools, signsrch [82], file [50], and Binwalk [77]. There exist some types of firmware that have no underlying operating system. They consist of only one binary file that operates directly on the hardware. In some cases, there is no abstraction of the OS and libraries, and in other cases, firmware images are not standard and no documentation is provided. Therefore, initializing a run-time environment and loading the binary is more challenging [140].

2.2 Binary Analysis Approaches

Binary analysis can be performed by inspecting the code statically, by executing it dynamically or by providing some symbolic values, which are called *static analysis*, *dynamic analysis* and *symbolic execution*, respectively. In the following, we review each of these approaches and their applications to the analysis of firmware images in embedded systems.

2.2.1 Static Analysis. Static analysis examines the code of a given binary program rather than executing it. It is typically designed to reason about the entire program, and has the capability to explore all potential execution paths of a given code. Static analysis techniques usually suffer from the C1, C2, C3, C4, C5, C6, C7, C8, and C9 limitations. For instance, they identify non-vulnerabilities which lead to high false positive rates [139], or they cannot find all the vulnerabilities (e.g., run-time vulnerabilities) that generates more false negatives. Additionally, since the information to trigger the identified vulnerability is not provided, the results of vulnerability detection should be verified manually. Static analysis approaches are scalable compared to dynamic analysis approaches; however, since they have their own limitations, the researchers recently tend to integrate these two approaches as outlined in Sections 3.2.1 and 3.2.3.

2.2.2 Dynamic Analysis. Dynamic analysis is the process of examining and monitoring the program behavior while it is running. Existing dynamic analysis approaches can be categorized as follows:

- *Fuzzing*: Fuzzing is performed by repeatedly generating malformed inputs randomly or based on the specific rules suitable for the target software, where the input will be processed incorrectly and consequently the program triggers an unintended behaviour that helps to identify the existence of vulnerabilities. In general, fuzzing approaches suffer from the C10 limitation. Applying fuzzing in embedded systems is more challenging than in tradition computing environment. For instance, memory corruption detection is more limited during fuzzing campaigns due to missing fault detection techniques (C11). Since the fuzzer cannot use the source code to derive memory semantics, detecting faulty states becomes more challenging. Hence, observing faulty states of the embedded system as part of a fuzzing campaign is the last resort. An existing work [114] shows that liveness checking is insufficient to capture various vulnerability types in the absence of source code, when the memory is corrupted.
- *Instrumentation*: Instrumentation is a technique to collect and insert execution feedbacks (e.g., system calls, run-time information, and crash harness functionalities) to analyze the behavior of the targeted application during the execution. Instrumentation information may be obtained at compile-time if the source code is available, or at run-time. Since the source code is rarely available, run-time instrumentation can be employed to re-write the binary and inject instrumentation information. However, run-time instrumentation is currently available only for embedded systems that have operating system and are compatible with current emulation. Instrumentation approaches suffer from C5, C10, and C12 limitations.
- *Dynamic Taint Analysis (DTA)*: DTA is a program analysis technique to inspect application bugs and vulnerabilities. It basically identifies the dependencies between the program inputs and its logic. As a result, DTA can detect improper input validation vulnerabilities resulted from the absence of sanitation checks on critical inputs. DAT also can be used to analyze the information flows in a binary code and to help automatic program test case generation. However, it suffers from C9, and C12 limitations.
- *Emulation-based techniques*: Emulation-based techniques build partial/full simulation for a specific architecture/platform, and then run the software within the simulated environment by employing powerful and advanced dynamic analysis techniques. Partial or full embedded system emulation is challenging due to vendor restrictions on documentation and development environment setup. The prominent initiatives in this area have the following shortcomings: limited CPU architecture support, no support for generic interrupt handling, limited number of peripheral modeling support [160], and being very slow [114]. Therefore, emulation approaches have C5, C9, and C12 challenges.

In summary, dynamic analysis techniques can overcome some limitations (e.g., C2, C3 and C7) of static analysis approaches. However, dynamic analysis techniques have the C5, C8, C9, C10, C11, and C12 common limitations.

2.2.3 Symbolic Execution. Symbolic execution techniques aim at reaching a specific program state by generating the inputs that satisfy the required path constrains. Instead of using concrete values as in dynamic analysis, the inputs are symbolic values. Hence, symbolic execution can explore all potential paths compared to concrete execution, which explores only one path related to the supplied concrete inputs. However, symbolic execution suffers from reliance on computationally expensive solvers (e.g., [15]) and path explosion (C8 and C9). Employing symbolic execution directly on firmware images is challenging for the following reasons: (i) Symbolic analysis requires a customized configuration based on the assumptions on memory layout and the location of memory-mapped hardware; (ii) Firmware intensively interacts with external environment via I/O peripherals. These behaviors are unknown to the analyst and difficult to model. Moreover, simulating the behavior is a challenging task as certain related information is only available to the vendors; (iii) The event-driven programming model of firmware regularly leads to infinite loops. As such, using heuristics to avoid possible infinite loops due to interrupts, may decrease the search space coverage. In contrast, tracking all possible paths leads to state space explosion; and (iv) Symbolic execution suffers from issues similar to those of

fuzzing (C11 and C12) as mentioned earlier [114]. When using symbolic execution, state exploration will proceed until a crash is detected or the termination condition is met, otherwise the execution will continue. Thus, if corruptions are not detected, symbolic execution spends the time on worthless states. Therefore, symbolic execution also has C11 and C12 limitations.

2.3 Taxonomy of Application Domains

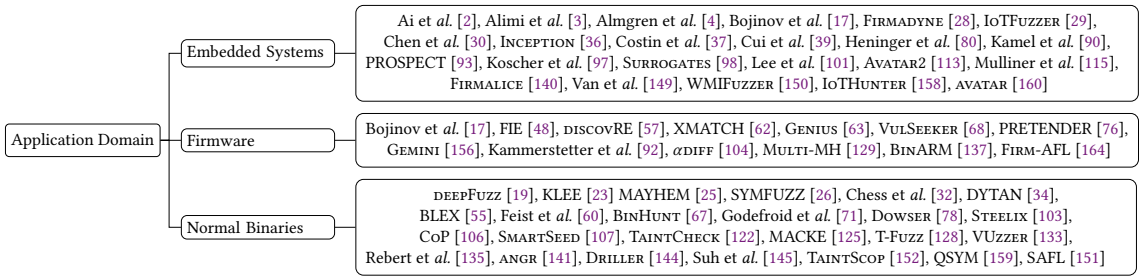

Fig. 1. Taxonomy of vulnerability detection domains

In this subsection, we categorize the existing works according to their application domains. Embedded system is the combination of hardware and sufficient software to perform the desired functionality, while firmware is the code embedded in a hardware which that code may not be necessary sufficient to perform the entire functionality of a device. Therefore, we group the approaches which are only focused on the analysis of the firmware into the 'Firmware' category, while those that consider any interaction with the hardware, are categorized as 'Embedded Systems'. Therefore, we categorize the existing vulnerability detection works into embedded systems, firmware, and normal binaries domains as presented in Figure 1. As can be seen, 38% of the works have been applied on the embedded system and 21% on them on firmware images, which highlights the importance of proposing novel solutions applicable to this domain.

2.4 Embedded System Classification

As explained earlier, an embedded system is composed of hardware and software designed to solve a specific task which may interact with its environment through sensors and actuators. A wide range of devices, such as digital cameras, printers, hard disk controllers, smartphones, automobiles, PLCs, and smart meters that can be considered as embedded devices. Embedded systems could be classified based on different criteria, such as the domain of usage, computation power, cost, and size. In order to preform a comparative study among existing works, we aim at classifying the embedded devices. Since the focus of this survey is on vulnerability detection in embedded systems, we follow the classification proposed by [114], which is based on the type of the operating system (OS). This classification can provide more information about the security mechanisms provided by a given embedded device compared to the aforementioned classifications (e.g., computation power). Based on this classification, embedded systems are divided into three types defined as follows:

- *Type-I: General purpose OS-based.* It generally represents embedded devices that utilize a modified lightweight version of Linux operating system to handle complex logic with access points, and routers.
- *Type-II: Embedded OS-based. Type-II* OSs are designed for embedded system devices with low computation resources, such as CPU, memory, or power. *Type-II* OS may not have a Memory Management Unit (MMU) as with

Type-III OS, however, the logical separation between applications and kernel still exist. Such OSs are generally accommodated in single-purpose embedded devices, such as LTE modems or DVD players.

- *Type-III: Embedded system without an OS-abstraction. Type-III* devices do not have an operating system, instead they use a single control loop and interrupt handler to response to events from the outer world.

3 DYNAMIC ANALYSIS

In this section, we provide a detailed review of dynamic analysis approaches and their applications to embedded systems. Our review will be organized around the taxonomy shown in Figure 2, which divides dynamic analysis approaches into three categories, i.e., *re-hosting*, *fuzzing*, and *dynamic taint analysis*. The dynamic analysis approaches are further categorized into two groups based on their application domains (i.e., embedded systems and normal binaries).

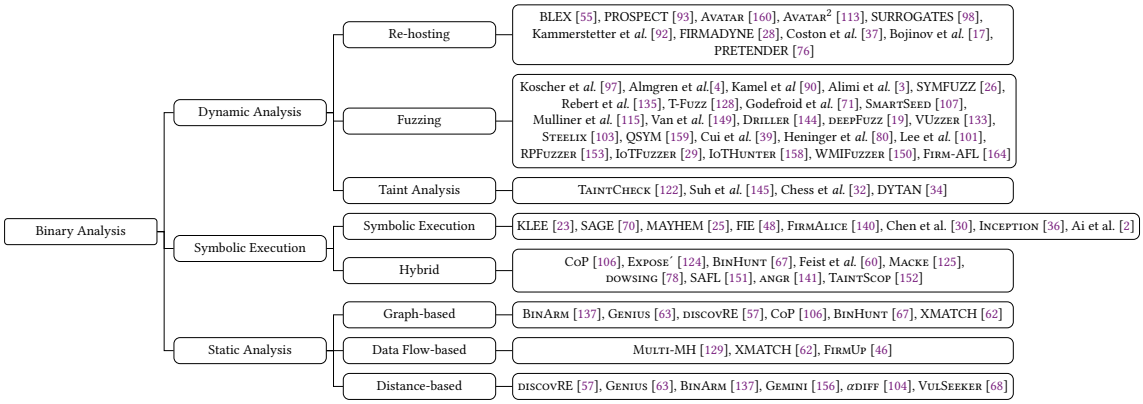

Fig. 2. Taxonomy of binary analysis approaches [Should be updated, I update symbolic execution section]

3.1 Dynamic Analysis over Embedded Systems

In this subsection, we review the works that employ re-hosting, fuzzing and symbolic execution on embedded systems.

3.1.1 Re-hosting. Embedded systems encounter specific scalability issues compared to regular binaries. In a traditional computing environment, parallel techniques using multi-processing or virtualization can accelerate testing campaigns. On the contrary, this is not practical in embedded systems since they require access to a batch of similar physical devices, which is impractical due to financial cost or limited resources (e.g., power supply and space). Moreover, testing an embedded system requires frequent restarts to ensure a clear state for each new test case. To tackle these issues, re-hosting techniques are proposed, where the firmware is extracted and moved to be executed in a different environment than its original one. In this subsection, we outline the existing contributions that perform partial/full emulations and web interface extraction using re-hosting.

Partial Emulation. Analyzing embedded systems in a virtual and scalable environment is a challenging task since running firmware, every so often, requires access to its related peripherals, which would not be found. Also, it is hard to emulate embedded peripherals due to lack of documentation or modeling complexity. To solve these issues, partial emulation is proposed where firmware is extracted and modified to be executed inside an emulator while forwarding commands to peripherals when it is necessary. Partial emulation gives the impression of having a full emulation of

the targeted embedded system with all its peripherals. However, its scalability hardly gets affected by the frequent interaction with the connected peripherals. Moreover, Scaling across multiple devices for multiple simultaneous testing can be a problem.

To facilitate using dynamic binary analysis approaches over embedded system devices, which need frequent access to their peripherals by its firmware hosted outside its original device, Kammerstetter et *al.* propose a partial emulator called PROSPECT [93]. PROSPECT implements a proxy to tunnel every peripheral hardware access initiated from the firmware running inside the QEMU [54] virtual machine to the embedded device under the test. To test this solution, PROSPECT is evaluated over a commercial fire alarm system by running it on the environment setup, and then an extensive fuzzing is carried out. As a result, a zero-day vulnerability is discovered on the targeted devices. PROSPECT cannot be applied to the devices that do not have network connection interfaces.

Similar to PROSPECT [93] but more scalable, a dynamic analysis framework named Avatar [160] is proposed to integrate actual hardware with a generic processor emulator. First, Avatar extracts the firmware images from the target device, and then injects the extracted firmware with software proxies. Therefore, it can run the extracted instructions inside the emulator while intercepting all the I/O operations to be forwarded to the physical device. Avatar is evaluated on three security scenarios: reverse engineering, backdoor detection, and vulnerability discovery on three different devices (GSM device, Hard disk, and wireless sensor nodes). However, it is evaluated only on ARM-based embedded systems, and needs more effort to be adapted to multiple CPU architectures.

In addition to partial emulation of embedded system as well as orchestrating the execution with physical embedded system peripherals, Avatar^2 [113] (a follow up work of avatar) can orchestrate interactions with other physical devices, debuggers, and popular binary analysis frameworks, such as QEMU, angr [141], PANDA [131] and OpenOCD [53]. Avatar^2 currently supports x86, x86-64 and ARM architectures. Its modularity feature makes it easy to adapt it to support additional architectures or even to implement an intermediate representation.

Since the emulation process is slow, general emulation solutions are not suitable for embedded devices which require near-real time response. To address this issue, a partial emulator called SURROGATES [98] is proposed for arbitrary ARM-based embedded systems to facilitate advanced dynamic analysis. SURROGATES can handle *clock changing*, *direct memory access*, and *interrupts*, and can also emulate the embedded device in near-real-time. SURROGATES utilizes a customized FPGA low-latency hardware to bridge the embedded system with the PCI Express bus of the host, which makes it faster than avatar [160] and more suitable to test time-sensitive and complex systems, such as medical devices.

In contrast to the aforementioned emulation approaches, Kammerstetter et *al.* [92] propose program state approximation and peripheral device communication caching between the running firmware inside a virtual machine and the targeted embedded device. First, the system learns the accessed peripherals behaviors by getting efficient training with a connected device. Then, the physical devices are no longer needed for further and repeated testing. Consequently, advanced dynamic analysis can be applied since this approach offers snapshotting and parallelization. However, the authors have shown that their peripherals caching approach can only be applied to small complex firmware. They also prove that the proposed approach suffers from state explosion similar to symbolic execution.

Full Emulation. To use dynamic analysis over embedded devices in scalable manner as with traditional computing devices, practitioners should have the capability to re-host targeted embedded system devices outside their device either in an emulation or in a virtualized environment without the need to have access to any peripherals. Full emulation is practical when embedded system is Linux-based and its firmware is successfully extracted [37], or when the targeted embedded system has a full available hardware documentation. Full emulation is more scalable than partial emulation,

and it does not require the physical existence of the targeted embedded system hardware. However, it is only applicable when peripherals of the targeted system can be successfully emulated. Full emulation opens the doors to introduce scalable dynamic analysis approaches into the embedded system world.

To achieve this goal, Chen et *al.* introduce a full-emulator dynamic analysis framework named FIRMADYNE[1] [28]. It is specialized in Linux-based firmware embedded system. FIRMADYNE is designed to be independent from the physical hardware and hardware-specific peripherals. Furthermore, it provides the capability to write to non-volatile memory and files which are generated dynamically. It utilizes QEMU [54], a full system emulator, to run a general Linux instrumented kernel with the file system extracted from the target firmware image. To evaluate FIRMADYNE, the authors collect a large datatset consisting of 23, 035 firmware images, and successfully extract file systems from 9, 486 firmware images. FIRMADYNE discovered 14 unknown vulnerabilities in 69 firmware images.

To make dynamic analysis (e.g., fuzzing, symbolic execution), more scalable and applicable on embedded system firmware running in a full-emulation or virtual environment, Gustafson et al. [76] propose PRETENDER framework, which automates the re-hosting of the various embedded systems' firmware in a virtual environment. Thus, dynamic analysis solutions can be performed in parallel similar to the traditional desktop computing environment. PRETENDER works as follows: First, it repeatedly runs the targeted firmware and records its real interactions with its related hardware. Then, records are used to build models for each available detected peripheral using machine learning and pattern recognition techniques. After that, the resulting models are integrated either with a well-known system full-emulator (e.g., QEMU) or integrated in a framework analysis (e.g., ANGR [141]) for further interactive and precise analysis in a scalable manner for its related firmware. PRETENDER was evaluated on 6 distinct "blob" images on three different platforms where each target includes synthetic security vulnerabilities. After that, each target are tested by using naive fuzzing and code coverage.

BLANKET EXECUTION [55] fingerprints binary code functions by creating binary function signatures. It executes the functions under controlled randomized environment and collects their execution side-effects as a signature. Consequently, functions with the same execution side-effects are considered to be similar. BLANKET EXECUTION is resilient to compiler optimization options; this capability radically changes the CFG structure by changing the number of nodes and edges. However, it is evaluated only on a single architecture (x86-64), and it is not scalable for a very large dataset.

Web Interfaces Analysis. For convenience, embedded devices, such as routers and switches, provide a web interface to be configured through or to be used to interact with the external environment. However, these web interfaces are vulnerable to various security threats. Coston et *al.* [37] propose a scalable and automated dynamic analysis framework to discover vulnerabilities in firmware that use web interface. The proposed system evaluates the security of web interfaces regardless of the device vendor. To this end, it initially extracts the embedded web server from the firmware, then it applies static analysis using RIPS [41]. Afterwards, it runs the web interface in an emulated device. Finally, the running web interfaces are tested using the open-source web penetration testing Arachni, zed attack proxy (ZAP), and w3af tools. The proposed work [37] is the first work that studies the security of PHP in embedded web interfaces.

FIRMADYNE [28] collects Linux-based firmware and runs them in the QEMU emulator. If the emulated firmware has a web access interface, web penetration testing is performed on the accessible web interface over a local network to check for well-known vulnerabilities, such as command injection, buffer overflow, and information disclosure.

Unlike the aforementioned approaches, Bojinov et *al.*[17] investigate the embedded systems web interface manually. They demonstrate that all the 21 investigated embedded system devices contain critical web vulnerabilities, e.g., cross

[1]https://github.com/firmadyne/

channel scripting (XCS). XCS utilizes multiple features included in modern embedded devices, such as cross-channels interactions between web interface and FTP server on network-attached storage (NAS) or the interaction between the web server and the and SIP phone service to compromise the embedded devices web interface.

3.1.2 Symbolic Execution. Symbolic execution is another perspective which has been exploited to tackle embedded system hardware compatibility *(C5)* and their peripherals modeling difficulties in particular. Therefore, instead of exporting firmware to be executed inside an emulator or virtual machine, the firmware will be executed symbolically inside a symbolic execution engine while treating peripherals access outcomes as symbolic data. Using symbolic execution improves scalability and help with exploring more code paths. However, it brings all symbolic execution limitation with it, such as path explosion. Path explosion could be worse when firmware require access to its peripherals. For instance, interrupt generated from the peripherals frequently creates more states [13].

FIE [48] is one of the early proposed approaches that introduce symbolic execution to embedded system world. It identifies the bugs related to analyzing memory safety of firmware running on the microcontroller family MSP430, which is employed in many critical security applications. FIE is built on a modified version of KLEE [23] symbolic execution framework to cover all the possible execution paths and to handle loop coverage via incorporating state pruning and memory smudging. FIE anticipates and describes the peripherals symbolically using only symbolic execution. FIE successfully identified 21 distinct bugs, and its scalability is demonstrated by a system built on Amazon EC2[6].

A cross-architecture framework called Firmalice [140] investigates the existence of variance authentication bypass vulnerabilities commonly known as *backdoors* in complex embedded systems firmware regardless of how the authentication mechanism has been implemented. To reason about the existence of *backdoor*, Firmalice builds a model based on the *input determinism* concept. It declares that any execution path derived from the entry point of the firmware to a privileged operation should go through a solid input validation process. Therefore, an attacker cannot bypass by means of information retrieval from the firmware image itself. To this end, Firmalice initially utilizes static analysis to extract the program data dependency graph, and then extract the program slices leading from the entry point to privileged operation location determined by a security analyst. Then, it employs its symbolic execution engine, inspired by KLEE, MAYHEM, and FuzzBALL [10], to find possible successful paths that lead to the desired privileged location. If the previous stages succeed, it will terminate with the required inputs needed to bypass vulnerability. Firmalice reported the existence of a backdoor in two commercial firmware devices.

Inception [36] is another symbolic execution framework for testing embedded system firmware. It is build in top of KLEE, which requires access to the source code. It solves the interrupts issues, resulted from the interaction with peripherals, by sending them into the symbolic engine in a tightly synchronized manner. To ensure real-time forwarding between memory access and real hardware, Inception framework introduce JTAG debugger. Moreover, it include a translator to lifted source code using LLVM and merge it with potential handwritten assembly instructions by developers. Inception has been evaluated over both synthesise and real word system, and it was able to uncover eight crashes, and two unknown vulnerabilities.

Concolic Execution. Concolic execution techniques integrate symbolic execution with concrete execution. Applying concolic execution over embedded systems is a challenging problem because it requires heavy computation resources such as CPU and memory, which embedded systems generally do not have. Moreover, symbolic execution requires instrumentation tools and theorem provers, which may not be supported by the targeted embedded system. To solve these issues, Chen et *al.* [30] propose to carry out the concrete execution on the targeted embedded system while performing the heavy-weight symbolic execution on a powerful remote host such as personal computers, workstations,

etc. To establish this coordination, cross-debugging functions, provided the embedded system vendors (VxWorks), are utilized, which make this work limited to VxWorks. Moreover, the proposed tool creates a one-to-one relationship between the symbolic execution engine and the targeted embedded device. Therefore, it affects its scalability.

Similarly to [30], Ai et *al.* [2] introduce concolic execution into embedded system testing using in same methodology in [30] while supporting multi-architecture such as *X86, ARM, and PPC*. To achieve this objective, the authors utilize a portable instrumentation scheme. They lift firmware and state information into an intermediate representation (VEX-IR) to make it easy for the symbolic engine to deal with constraints in multiple CPU architectures.

3.1.3 Fuzzing. Re-hosting the firmware outside its environment has several obstacles, as mentioned in *C5, C9, C12*. To overcome these challenges, several approaches suggest testing targeted embedded systems over a communication interface. In the following, we outline these solutions. Chen et *al.* [29] propose IoTFuzzer, which leverages vendor's mobile apps given their capabilities to remotely control their related devices. IoTFuzzer performs dynamic analysis on the target firmware mobile app to derive the logic of the device command messages. This allows identifying command format, sent URL, and used encryption schemes. Then, data-flow analysis is used to instruct the app to issue meaningful test cases by altering the content of learned messages. The generated test cases fuzz the target device firmware to uncover memory corruption vulnerabilities. This enables a guided protocol-based black-box fuzzing with no particular protocol specification, which works even on proprietary communication protocols. To monitor crashes resulted from the fuzzing, the status of the targeted device are checked by sending a message that query the status of the targeted device. In contrast, AutoForge [165] only works with standard protocols (e.g., HTTP) and cannot deal with custom designed cryptography functions. IoTFuzzer was evaluated on 17 real-world IoT devices and found 15 vulnerabilities, eight of which have never been reported before. However, IoTFuzzer cannot provide the location of a vulnerability but only the input that triggers it.

While IoTFuzzer shows promising results over fuzzing IoT firmware, it may unsuitable for firmware implementing complex stateful messaging protocols (e.g., SNMP, FTP, SSL, BGP, SMB), which maintain strict validation mechanisms (e.g., checksum, message length, protocol ID) for almost every received message. Hence, malformed inputs will easily diverge from the current protocol state, and thus miss the chance of discovering deep bugs. To address stateful network protocols, Yu et al. [158] propose IoTHunter which implements a new technique (multi-stage message generation). IoTHunter is integrated with AFL, Boofuzz [18], and Avatar2. Along with its capability to fuzz known states, it can also explore unknown states from a given state sequence. To achieve feedback-based state exploration and perform coverage-guided grey-box fuzzing, it shifts to another protocol's state based on its given state sequence. IoTHunter was evaluated on eight real-world IoT programs from home router Mikrotik, and uncovered five new vulnerabilities. It also provides better results than the black-box fuzzer Boofuzz [18] with respect to edge coverage, block coverage, and function coverage.

Similarly to IoTFuzzer, WMIFuzzer [150] uses alterations and fuzz running IOT firmware without the need of a predefined data model. WMIFuzzer applies fuzzing on the web management interface of Commercial off-the-shelf (COTS) IoT devices for administration or user interaction. However, there is no unified specification to be followed when designing such an interface. To address this, WMIFuzzer first constructs the initial legitimate message seeds (compatible with the targeted COTS IoT device) using UI automation and exhausting all possible GUIs. Then, the valid GUIs captured by the applied proxy will be used as seeds. Second, the captured messages are converted into Abstract Syntax Trees where the message's fields contents are stored within the trees' nodes. Thus, alteration will be applied only on the tree nodes to ensure valid message construction. The final step is performing the fuzzing over the altered

messages. Network monitoring techniques are implemented to detect if a vulnerability was triggered. Since not all triggered vulnerabilities will crash the target device, the altered message will contain injected reboot command and Interface Leak. Then, their side effects are monitored from the observed network. WMIFuzzer was evaluated on seven COTS popular IoT devices, and it discovered 10 vulnerabilities, of which 6 are 0-day ones. The limitation of WMIFuzzer is that is works only with IoT devices that have a web management interface accessed over either HTTP or HTTPS.

RPFuzzer [153] was proposed prior to IoTFuzzer and WMIFuzzer. It introduces a predefined expert crafted mathematical model on the protocol to generate the initial messages (seeds) to be used later as a start point during the fuzzing campaign. Then, mutation-based fuzzer is applied on the seeds for test case generation. To detect if a vulnerability was triggered, three monitoring methods are used: sending normal packets, monitoring CPU utilization, and checking system logs. RPFuzzer was evaluated on Cisco routers where experiments are launched over the SNMP protocol. RPFuzzer discovered eight vulnerabilities, five of which are 0-day.

Zheng et al. [164] propose Firm-AFL for Gray-box fuzzing on embedded systems to improve over gray-box fuzzer such as AFL, which does not work with various embedded devices due to compatibility issues. Firm-AFL is built on top of Firmadyne [28] as the latter already solved many hardware compatibility issues compared to other emulators such as Avatar [160]. Firm-AFL is also solving fuzzing throughput issues over full-emulation environment by fuzzing the targeted firmware program in a user-mode instead of system-mode, whenever system mode emulation is not required.

To evaluate the security of embedded devices connected to the internet, Cui et *al.* [39] perform global network scanning on both telnet and HTTP ports using Nmap to detect firmware devices with the default passwords provided by their vendors. The results of four-month scanning indicate that about 540, 000 devices are accessible using their default passwords. Similarly, Heninger et *al.*[80] perform network scanning on TLS and SSH servers. The authors demonstrate that the majority of embedded system devices are using weak RSA generation key algorithms that have low entropy. Furthermore, it is shown that some embedded systems are using predictable DSA private keys.

Alimi et *al.* [3] propose a mutation-based fuzzing approach using genetic algorithm to generate test-inputs for examining the Master Card for vulnerabilities or abnormal behaviors. Furthermore, the authors propose a method to evaluate the results of the generated inputs to optimize the search process of finding commands that lead to unwanted behavior by the Master Card specification. The authors observe that the tested cards could be corrupted due to intensive fuzzing testing; consequently, they examine the targeted part in a simulation environment. Finally, they successfully trigger an acceptance of prohibited transactions, and can identify the context that lead to these illegal transactions.

Some smart cards implement web technologies to facilitate the communication with other networks. Kamel et *al.* [90] implement a *black box* based fuzzer to investigate the potential vulnerabilities of web servers which are embedded in smart cards. They further test whether the implemented HTTP server compliance with the design specification. The evaluation results demonstrate that some smart card HTTP severs accept administration commands that should be available only to the card issuer. Furthermore, several non-compliant behaviors have been observed.

Fuzzing has been applied to investigate potential vulnerabilities in the modern automobile systems. Modern automobiles are controlled by the various computerized and networked system. Koscher et *al.* [97] performed a comprehensive investigation over two 2009 automobiles in a controlled lab and real street. They implemented a system capable to sniff and fuzz the packets sent into the vehicle CAN bus. They successfully generate packets to perform various functionalities, including unlocking doors, triggering the horn, turning off the light, etc. They also demonstrate crafting attacks capability that can endanger the driver and passengers (e.g., suddenly disengaging the brakes while driving).

Similarly, Lee et *al.*[101] propose an approach to randomly generate fragments of CAN packets, which can be applied over different CPU architectures. They observe clear alterations in the instrumentation panel of the targeted car. Fuzzing

also has been utilized to test PLCs and smart meters for potential vulnerabilities. Almgren et *al.* [4] implement both mutation and random fuzzers to examine PLCs and smart meters. The experiments demonstrate that the generated input packets are able to uncover potential denial of service vulnerabilities, as well as corrupt the PLC execution.

To test smart-phones adherence to the GSM specification, Mulliner et *al.* [115] and Van et *al.* [149] design a generation-based fuzzer that can trigger reboot, memory exhaustion and denial of service.

3.1.4 Comparative Study.

Due to the absence of similar setup environment, embedded systems and firmware images, a comprehensive evaluation of the existing solutions is not feasible. However, we conduct qualitative and quantitative comparisons based on the available information provided in each solution in terms of the approaches, implementations and evaluations as demonstrated in Table 1 and Table 2.

Approach	Venue	Methodology								Target		Device Type			CPU Arch.				Release	
		Partial Emulation	Full Emulation	Symbolic Execution	Fuzzing	Web Interface Check	Network Scanning	Taint Analysis	Static Analysis	Firmware	Embedded Systems	I	II	III	x86-64	ARM	MIPS	Reused Framework(s)	Open Source	Open Service
Koscher et *al.* [97]	S&P 2010				•						•			•	•	•	•			
Cui et *al.* [39]	ACSAC 2010				•		•				•	-	-	-	-	-	-			
Mulliner et *al.* [115]	USENIX 2011				•						•		•		-	-	-			
Heninger et *al.*[80]	USENIX 2012				•		•				•	-	-	-	-	-	-			
Kamel et *al.* [90]	IJINS 2013				•						•	•			-	-	-			
FIE [48]	USENIX 2013			•						•			•	•				KLEE		
RPFuzzer [153]	TIIS 2013				•						•		•		•	•	•			
Almgren et *al.* [4]	CRISALIS 2014				•						•		•			•				
Van et *al.* [149]	ESSoS 2014				•						•		•		-	-	-			
PROSPECT [93]	ASIACCS 2014	•								•	•	•				•	•	QEMU		
Avatar [160]	NDSS 2014	•								•	•		•	•		•		QEMU	•	•
Alimi et *al.* [3]	HPCS 2014		•		•						•			•	-	-	-			
Chen et *al.* [30]	TR 2014			•									•	•						
Lee et *al.*[101]	AINA 2015				•						•			•	-	-	-			
Firmalice [140]	NDSS 2015			•					•	•	•	•	•		•	•	•	KLEE, MAYHEM,FuzzBALL		
SURROGATES [98]	WOOT 2015	•								•	•	•		•		•				
FIRMADYNE [28]	NDSS 2016		•							•	•	•				•	•	QEMU		
Kammerstetter et *al.* [92]	SECUWARE 2016	•								•	•	•					•	QEMU		
Costin et *al.* [37]	ASIACCS 2016		•							•	•	•			•	•	•		•	•
Avatar^2 [113]	BAR 2018	•								•	•		•			•	•	angr,PANDA	•	•
Inception [36]	USENIX 2018			•									•	•		•		KLEE,LLVM,JTAG		
IoTFuzzer [29]	NDSS 2018				•			•			•	•	•	•	•	•	•			
IoTHunter [158]	CCS 2019		•		•						•	•	•		•	•	•	AFL,Boofuzz,Avatar2		
WMIFuzzer [150]	SCN 2019				•	•					•		•		•	•				
Firm-AFL [164]	USENIX 2019	•	•		•					•		•				•	•	Firmadyne, AFL	•	
PRETENDER [76]	RAID 2019		•							•		•				•	•	ANGR,QEMU		
Ai et *al.* [2]	ICCSP 2020			•						•	•	•			•	•		SE, VEX-IR	•	
Distribution	NA																			

Table 1. A comparison of state-of-the-art embedded system vulnerability detection approaches.
(•) means that the approach offers the corresponding feature, otherwise it is empty. (–) means that the information is not provided.

The first and second columns in Table 1 specify proposals and corresponding venues ordered by the date. The next eight columns present their main proposed methodologies. The 'Target' column specifies if the outlined target is running embedded device, re-hosted firmware, or both. The 'Device Type' column specifies the class of the embedded devices tested by a given proposal. In next three columns, we mark which CPU architectures are supported by these approaches. It is worth noting that only one work (FIE [48]) is running on the microcontroller family MSP430, which we have not

included in this Table. Moreover, the work proposed by Bojinov et *al.* [17] checks the web interface vulnerabilities manually, therefore the type of CPU architecture is not important. The next column indicates which framework(s) have been used by the approaches. Finally, the last two columns show which tools are open source and which ones are accessible as a service to public to examine their code.

Moreover, more details on these approaches is provided in Table 2. The first column of Table 2 specifies proposals. The 'Challenge Overcome' column outlines the type of challenges that a given proposal has tackled. The 'Target (sector)' column reports the domain field of the targeted tested embedded device. The next two columns specify if a given proposal needs access to the source code of the firmware of the embedded device and the physical device itself, respectively. The last column ('Protocol') outlines the type of communication protocol(s) used to test the targeted embedded device by a given proposal.

Approach	Challenge Overcome	Target (Sector)	Access		Protocol
			Source Code	Device	
Koscher et *al.* [97]	C5,C10,C11	Automotive			CAN
Cui et *al.* [39]	C5,C9	IoT devices		•	Telnet, HTTP
Mulliner et *al.* [115]	C10,C11	GSM feature phones			
Heninger et *al.* [80]	C5,C9	IoT devices		•	TLS
Kamel et *al.* [90]	C5,C11	Smart cards			
FIE [48]	C5, C9	Micro-controller FamilyMSP430	•		
RPFuzzer [153]	C5,C10,C11	Cisco routers		•	SNMP
Almgren et *al.* [4]	C5,C11	PLCs and smart meters			
Van et *al.* [149]	C10,C11	GSM feature phones			
PROSPECT [93]	C5,C9	Building automation (alarm system)		•	TCP/IP
Avatar [160]	C5,C9	GSM Phone, Hard Disk, ZigBee sensor		•	
Alimi et *al.* [3]	C5,C10,C11	Master Card		•	HTTP
Chen et *al.* [30]	C5,C9,C10	VxWorks		•	
Lee et *al.*[101]	C5,C11	Automotive			CAN
Firmalice [140]	C5,C9,C10	Smart meter, CCTV camera, Dell Laser Mono Printer			
SURROGATES [98]	C5,C9	Medical devices		•	
FIRMADYNE [28]	C5, C9, C11	Embedded system with network/web services (e.g., routers)			HTTP/HTTPS, Telnet
Kammerstetter et *al.* [92]	C5,C9	Simple firmware (GNU core utilities)		•	TCP/IP
Costin et *al.* [37]	C5,C9,C11	Routers and switche			HTTP
Avatar^2 [113]	C5,C9	PLC, Firefox		•	
Inception [36]	C5,C9,C10	Industrial applications, boot loader	•	•	
IoTFuzzer [29]	C5, C10,C11	Smart home devices (e.g., router, printer, IP camera)		•	HTTP/HTTPS, UDP/TCP
IoTHunter [158]	C5,C9,C10,C11	Home router and NSA Synology			NMP, FTP, SSL, BGP, SMB
WMIFuzzer [150]	C5,C10,C11	SOHO router, IP camera, gateway		•	HTTP/HTTPS
Firm-AFL [164]	C5,C9,C10,C11	Routers			HTTP, SSH
PRETENDER [76]	C5, C9	ARM mbed			
Ai et *al.* [2]	C5,C9,C10	FriendlyARM mini2440, Kyocera 8000 series printer, binutils programs		•	

Table 2. A comparison of state-of-the-art embedded system vulnerability detection evaluations. [C8?]
(•) means that the approach offers the corresponding feature, otherwise it is empty.

Discussion. One of the main objectives of variant binary analysis over embedded system devices is to overcome scalability issues, hardware compatibility, and making well-known dynamic analysis approaches over traditional environment applicable in the embedded system world. Narrowing the observation down, we can see that, proposed approaches are less likely to target embedded devices of type-I than the other approaches that target embedded devices of Type-I and Type-II. Moreover, fuzzing embedded systems through their provided network connection interface shows efficient performance by discovering more bugs and vulnerabilities with less effort compared to re-hosting approaches and symbolic execution. Furthermore, the majority of approaches do not need access to firmware source code except

with exception of two, since those works are based on the KLEE [23], which works over source code. Moreover, full emulation approaches or approaches integrated with full emulation firmware are more scalable because they do not need access to any hardware. In addition, re-hosting approaches try to overcome multi-CPU architecture challenge (C5), and scalability (C9). On the other hand, fuzzing approaches try to overcome in addition test case generation (C10), and fault detection (C11). Finally, Introducing AFL and concolic execution into an embedded system testing world open the door of introducing state-of-the-art variant approaches that have been proposed so far for a general-purpose computing environment.

3.2 Dynamic Analysis over Normal Binaries

In this section, we summarize and categorize state-of-the-art binary analysis approaches, which have been extensively evaluated on traditional computing systems, such as desktops. Although the main focus of these approaches is not embedded system devices, some of which have been already introduced or integrated into embedded systems (as mentioned in Section 3.1.1). For example, emulation frameworks are successfully integrated with the well-known binary analysis frameworks, such as ANGR [141] (black-box mutation fuzzer and white-box fuzzers) and AFL [164] (gray-box fuzzer). Moreover, Chen et *al.* [30] and Ai et *al.* [2] introduce the combination of symbolic execution and concolic execution into embedded system testing. These combined approaches significantly have improved the usability of dynamic analysis over embedded system devices.

In the following sections, we outline state-of-the-art approaches in the fuzzing, symbolic execution, and dynamic taint analysis approaches. Then, we provide a quantitative and qualitative comparison between these approaches. For further detail, We refer the reader for more details on the fuzzing and symbolic execution approaches applied on normal binaries to existing fuzzing surveys such as [108], and symbolic execution survey [13], respectively.

3.2.1 Fuzzing.

Fuzzing approaches can be divided into three categories of *mutation-based*, *generation-based* and *hybrid* as follows:

Mutation-based Fuzzers. Mutation-based based fuzzer are easy to implement and use. They have been utilized by many state-of-the-art fuzzers. Test cases are generated by using random mutation techniques. Random mutation techniques basically start with a prepared sample or seed. Then, in a repetitive manner, parts/bits of the input samples are mutated randomly or probabilistically. Next, the obtained samples are fed into the tested program/system while monitoring their behavior waiting for an error or a crash that might be triggered by one of the mutated inputs. In the following, we discuss existing works that employ random mutation.

American Fuzzy Lop (AFL) [161] is a simple and efficient coverage-based fuzzer. It has been widely used for detecting vulnerabilities. AFL is a grey-box Fuzzer. It utilizes coverage approach as a guidance to explore a given software. Also, it employs instrumentation to report branch coverage. Therefore, it records which software code regions have been explored and which ones have not. However, it cannot provide other information, such as which inputs are needed to explore non-visited code regions, or which parts of the supplied seeds are required to be mutated to help reaching non-visited code regions. AFL stores promising test cases in a queue and explores them in round robin manner. On every selected seed, it performs deterministic fuzzing, random fuzzing and splicing. Interesting seeds obtained from previous fuzzing, and reaching new branches, will be added into the queue [127].

Rebert et *al.* [26, 135] propose a mathematical solution to efficiently select seeds for any type of fuzzer within a constrained time and budget in order to increase the number of discovered bugs. First, scheduling the selection of the fuzzy seed is formulated as a linear programming problem. Then, a ground truth fuzzing dataset, consisting of different

file types formats, is collected. Next, different seed selection algorithms are implemented and evaluated on a subset of the dataset. It is shown that the algorithm selection choice increases the number of discovered bugs greatly. It is also demonstrated that seed selection based on a good coverage improves bug discovery process compared to random seed selection. Moreover, the authors show that in practice, it is more efficient to generate a reduced set of seeds, from the full dataset, to subsequently use in the fuzzing campaign. Furthermore, they outline that the generated reduced datasets are transferable among programs that accept similar file types as an input instead of generating input seeds for each individual program. As a result, 240 bugs across eight application are reported.

To maximize bug discovery in black-box fuzzing, Cha et *al.* proposed SYMFUZZ [26]. SYMFUZZ employs a probability model to calculate the mutation ratio from given a pair of program seeds and the number of found crashes. Moreover, SYMFUZZ utilizes symbolic analysis to work over the execution traces of the targeted program to optimize the parameters for the black-box fuzzer. As a result, the authors demonstrate that the mutation ratio can be inferred optimally from the dependence relations among the input bits of the targeted program. Moreover, they show that each program has different optimal mutation rates, which has a major influence during fuzzing. SYMFUZZ was used to test 8 Linux utilities, and it has discovered 110 new crashes. It discovered 39.5% and 57.9% more bugs compared to well-know *BFF* [81] and *zzuf* [2] mutational fuzzers that apply bit-flipping-based mutation, respectively.

Instead of mutating the input seed to cover more execution paths within the targeted program, T-Fuzz [128] transforms target programs to accept provided seed input by removing or disabling the sanity check depriving the provided seed from advancing in the execution path. This process is continued until bugs get discovered using light-weight dynamic tracing technique. Afterwards, post-processing symbolic execution is applied, which adjusts the seed input that had triggered the discovered bug on the transformed program, to suit the original program and to remove the discovered false positive bugs. T-Fuzz has been evaluated against DARPA CGC dataset [43], LAVA-M dataset [52], and four real-world programs. T-Fuzz showed its superiority over Driller [144] and AFL [161].

Grammar-based Fuzzers. Generation-based fuzzers need to know the input format of the targeted program to generate the testcases. Testcases made by traditional random mutation techniques are more likely to be rejected at the initial state of the program execution because they do not match with the required input format. Grammar representation techniques address the limitation faced by random mutation techniques by restricting generated inputs into a specific data structure or grammar to ensure reaching the deep level of a program. In the following, we discuss the initiatives grammar-based seed generation.

Godefroid et *al.* [71] propose a neural-network-based approach to learn the structured inputs of a program from samples. They generate well-structured, diverse, and high-coverage *input file seeds* to be later used to fuzz the targeted program. To this end, they apply recurrent neural networks *char-rnn* [72, 110] provided with reacquired knowledge on probability distribution of the learned model. After successfully generating valid inputs with the required structure, fuzzing is applied to temper the structure of generated seeds to increase the chance of reaching to unpredictable code paths where bugs may be encountered. The approach is evaluated by training the model with complex PDF format files. The generated inputs by the trained model have been used to fuzz *Edge PDF Parser*, and previously unknown buffer overflow vulnerabilities have been successfully discovered.

Similarly, SmartSeed [107] applies generative adversarial networks (WGAN) [9] to train a model from selected samples for generating seeds in multiple formats (e.g., mp3, bmp and flv). Moreover, SmartSeed can learn to generate corrupted seed files to be utilized to trigger more crashes. Furthermore, it has been designed and tested to be compatible

[2] http://caca.zoy.org/wiki/zzuf

with other fuzzers, such as AFL. SmartSeed has been tested against 12 applications; and it discovered 16 unique vulnerabilities and doubled the triggered crashes when compared to previously proposed approaches.

Hybrid Fuzzing Approaches. To address fuzzing approaches limitations, and increase the probability of locating more potential bugs, fuzzing approaches have been integrated with other binary analysis approaches to help with bypassing conditional branches checking either on a concert or magic value. To tackle the limitation of fuzzing, Driller is proposed, which combines both concolic execution and fuzzing and runs them in a repeated and alternate manner to locate vulnerabilities hiding in deep down level paths. Driller starts by fuzzing the targeted binary file until it gets stuck due to a complex conditional path check, such as comparison against concrete values or magic values, for a dedicated time. To bypass the conditional path wall, Driller utilizes a concolic execution constraint-solving engine to identify the inputs needed to pass conditional path walls check. Consequently, Driller updates the fuzzer with new inputs to continue the exploration, and it repeats this procedure until the crash occurs in the target application. Driller has been evaluated against DARPA CGC dataset, and locates the same number of vulnerabilities that the winner of the DARPA CGC contest achieved within 24 hours.

Similar to Driller, deepFuzz [19] combines concolic execution with constrained fuzzing. DeepFuzz gives probabilities to each path resulted from the symbolic execution solver, and the most likely paths will be further explored by the fuzzer. However, applying probabilities to each path and only visiting the path with high probabilities may result in non-visited paths. Consequently, potential vulnerability that reside at lower probable paths would be missed as well.

To reduce the cost of using symbolic execution, Wang et *al.* [151] propose SAFL, a dedicated fuzzer for C/C++ programs. SAFL uses symbolic execution only once to create good initial seeds and then continues with fuzzing guided by an efficient mutation algorithm. SAFL uses similar components used by Driller, except the oscillation between fuzzing and symbolic execution since it is costly. SAFL has been evaluated over 10 applications, and discovered 109% more crashes compared to FastAFL. SAFL has also the potential to reach rare deep locations on the tested program.

Even though combining concolic execution with fuzzing shows improvement in vulnerability detection as proposed by [144], it does not scale well with large complex applications and remains as a challenge [24]. Rawat et *al.* [133] state that the limitation on combining concolic execution with fuzzing is that, they are applied in application-agnostic manner. To address this limitation, VUzzer [133] is proposed, which combines fuzzing with dynamic taint analysis (DTA). To guide the fuzzer toward reaching the deeper paths and maximize the coverage, DTA with static analysis is utilized to extract data flow and control flow features. Therefore, magic values and concrete value comparisons are identified easily instead of using calculation intensive symbolic execution to solve the constrains. Moreover, weights are assigned to each basic block of the extracted control flow graphs, instead of treating them equally. As a result, generating interesting inputs for fuzzing are faster. VUzzer has been evaluated against CGC dataset and LAVA dataset, and it was able to trigger bugs with an order of magnitude less generated inputs compared to AFL. Moreover, VUzzer reduces the execution time for taint analysis compared to using pure fuzzing approach as in the case of AFL.

Similar to VUzzer [133] and Driller [144], Steelix [103] utilizes low-weight program analysis by performing both static analysis and binary instrumentation. Steelix utilizes static analysis to extract basic blocks and interesting comparisons instructions to be later used by applied binary instrumentation. During testing, binary instrumentation provides feedbacks to the fuzzer including both path coverage and comparison progress. The comparisons progress feedback notifies the fuzzer on how many bytes are correct within the supplied inputs. Therefore, the fuzzer will keep these bytes and invest more time on their neighbors. Steelix has been evaluated against LAVA-M dataset and DARPA CGC dataset in addition to five programs, and it was able to locate nine new bugs and one CVE.

Yun et *al.* perform a study on the approaches that combine fuzzing with concolic execution to identify the reasons behind their non-scalability [159]. They observe that the expensive calculations are not only due to path explosion, but also as a result of concolic execution emulation engine implementation. Moreover, they observe that applied concolic execution engine functions over lifted binary instructions (e.g., VEX-IR or LLVM-IR) is also computationally expensive. To address these issues, they re-implement the symbolic execution engine to work directly on native extracted instruction sets through utilizing dynamic binary translation, which results in major reduction in time complexity that current implementations suffer from. Finally, the newly concolic implementation design are combined with fuzzing in a similar manner to Driller. The proposed prototype is implemented in a system called QSYM [159]. QSYM has been evaluated on both DARPA CGC and LAVA datasets, and the evaluation results demonstrate its superiority over Driller and VUzzer, respectively. QSYM discovered 13 new bugs in eight real-world programs, such as Dropbox and OpenJPEG, which have been extensively tested before with state-of-the-art fuzzers, such as AFL, and OSS-Fuzz Google fuzzer.

3.2.2 *Dynamic Taint Analysis.*

In the following, we discuss various works on dynamic taint analysis (DTA). Clause et *al.* [34] propose a general taint analysis framework called DYTAN, which performs data flow and control flow taint analysis. It can be configured to choose which data objects are tainted and how the marked tainted data could be propagated. DYTAN can function on source code or stripped binary files, or with debugging symbols if available. In this work, two approaches are implemented, namely the *prevention of overwrite attacks*, and the *detection of SQL injection*. DYTAN is sensitive to specific characteristics, such as neglecting control flow taint related propagation. It is evaluated against the Firefox web browser to show its practicality.

On the other hand, Suh et *al.* [145] utilize DTA to protect programs from malicious inputs at operating system level. The authors propose a software module working at the operating system level. It marks every controlled users' input as malicious, and then it keeps track of their propagation. Therefore, during the execution process, based on the operation type, the CPU checks whether the results of the executed operation is different from what it is expected. This approach functions transparent to the to users as the application program run.

Chess et *al.* [32] apply DTA to locate input validation vulnerabilities by leveraging the calculation devoted to functional testing. The proposed approach works as the follows. First, user inputs are marked as tainted; then, the targeted program is monitored at runtime. If the tainted user input reaches a critical location without encountering any input validation, a potential vulnerability is reported. The authors state that their approach outperforms traditional security testings (e.g., fuzzing) for locating input validation vulnerabilities when the search space is large and complex.

Newsome et *al.* propose TaintCheck [122] to automatically detect overwrite attacks and generate signatures for them. TaintCheck identifies tainted data, such as network data, and keeps tracking these data along with their related propagation during program execution. Then, it reports when tainted data are used in sensitive memory locations, such as return addresses or format string. TaintCheck has no false-positive [122].

3.2.3 *Symbolic Execution.*

In this section, we first review dynamically performed symbolic execution approaches, and then review hybrid approaches that combine symbolic execution with other techniques (e.g., taint analysis).

Symbolic Execution Approaches. Symbolic execution is employed to maximize path coverage exploration to discover potential bugs and vulnerabilities. The popular symbolic execution tool called KLEE [23] generates test cases automatically with high coverage for both simple and complex software programs. KLEE aims to reach every executable

instruction and investigates critical operations that could be potentially exploited by attackers. To this end, KLEE implements different constraint solving optimizations (e.g., expression rewriting) to increase efficiency, decrease solving time and reduce memory consumption. KLEE maintains a compact program state through implementing *copy-on-write* at the object level, and it designs heap memory as an immutable map. As a result, more states can be represented and explored. To avoid potential path explosion, KLEE stops forking at each branch. Instead, it implements immutable state representation to preserve memory space capacity. KLEE has been evaluated over *GNU coreutils* and successfully reported 56 serious bugs in 452 tested applications. Also, it achieved an average code coverage of 90%.

To maximize path coverage exploration to discover potential bugs and vulnerabilities, especially over large and complex applications, a novel algorithm named *generational search* is proposed within a tool called SAGE [70]. SAGE works as the follows: it runs a targeted program with the well-formed input file and record its execution traces. Then, symbolic execution is applied on the recorded traces to create constraints that are obtained from the last execution of the program-input pairs. To maximize the input tests coverage, each generated constraint is negated at every conditional path in a one-by-one fashion. Then a symbolic constraints solver is applied to check their satisfiability. Thus, satisfiable constraints are reflected in the newly generated inputs to be examined against the targeted program where more priority is given to the inputs that have more path coverage. SAGE locates `MS07-017 ANI` vulnerability, which neither extensive black-box fuzzing nor static analysis approaches, have been able to find. Furthermore, SAGE has detected more than 30 new bugs over various complex Windows applications, including media players and image browsers.

Symbolic execution can be applied to dynamically craft the required inputs needed to exploit and run discovered vulnerabilities. To achieve these goals, MAYHEM [25] integrates both symbolic execution and concrete execution in an alternate manner to achieve a synergistic effect in terms of speed and memory utilization. MAYHEM introduces a memory-based indexing model to handle symbolic memory indexes at the binary level, which helps in discovering more vulnerabilities. MAYHEM is applied on 29 applications and successfully reports and generates 29 exploits automatically. However, it supports only a limited number of system calls for both Windows and Linux platforms, and it can only investigate single execution, in contrast to S2E [33] which supports multiple threads of executions, and it examines both user and kernel programs. MAYHEM is capable of locating standard vulnerabilities (e.g., stack overflow); however, it does not support finding sophisticated vulnerabilities, such as heap-based overflows and use-after-free.

Symbolic Execution Combined with Static Analysis. To combine symbolic execution and static analysis, Feist et *al.* [60] propose a system that integrates static analysis with symbolic execution to excavate and prove the existence of Use-After-Free vulnerabilities (UAF) in binary code. First, static analysis is used to locate the code slices susceptible to possible UAF vulnerability by utilizing value set analysis [12] implemented by GUEB [75]. Second, a guided symbolic execution is used to validate reachability from the entry point. This approach finds unknown vulnerability on Jasper application. However, it is limited to UAF vulnerabilities.

On the other hand, MACKE [125] integrates static analysis with symbolic execution to uncover buffer overflow vulnerabilities and report their severity scores. MACKE works as the follows: First, symbolic execution is employed on every function in the targeted binary separately to ensure full search coverage. Second, it utilizes information from static code and inter-procedural path feasibility to combine the results from each function to reason on their feasibility. Finally, it helps security analysts to make their final decision on fixing potential vulnerabilities based on their related risks and severity score. Even though this work is proposed to solve the issue related to search space coverage, path explosion is still possible if one of the functions is very big and contains a large number of complex paths. Furthermore, MACKE has been evaluated only on four open-source programs and not at large scale.

To facilitate the integration of various binary analysis approaches, a systematized open-source binary analysis framework named ANGR introduced [141]. ANGR reproduce a set of state-of-the-art binary analysis approaches implementations in a single, coherent framework. This framework provides the capability to directly compare different implemented approaches. Besides, it offers the capability of composing two or more approaches together in a manner that leverages each approach advantages and compensates for each approach limitations. Furthermore, ANGR re-implements some offensive binary analysis, such as automatic exploit generation, ROP shellcode, exploit replay, etc. ANGR lifts the binary program into VEX-IR intermediate representation to support variant CPU architectures.

Moreover, static similarity approaches have been also integrated with symbolic execution to identify vulnerabilities in normal binaries. For instance, BINHUNT [67] employs maximum common subgraph (MCS) isomorphism using backtracking algorithm [148] on the intermediate representation of assembly instructions to compare both CFGs and call graphs of two binary programs. Additionally, symbolic execution combined with simple theorem proving (STP) [66] is used to compare pairs of basic blocks. EXPOSÉ [124] combines semantic execution (using STP constraint solver) with the syntactic matching (using cosine distance of the function n-grams) to compare two functions for binary code re-use detection. BAP [22] is modified to process x86 instructions to be used with STP. A filtering process is used to prune the search space by both excluding compiler loader support functions and identifying improbable function pairs based on four attributes (e.g., number of input arguments). However, EXPOSÉ is too coarse-grained in some cases to identify vulnerable functions.

Symbolic Execution Combined with Taint Analysis. Generated malformed inputs by fuzzing tools are rejected usually at the early stage of the executing target program, due to strict sanity checks, especially when the target programs employ check-sum over given inputs. To overcome this, Wang et *al.* [152] propose TAINTSCOP framework which is an automatic checksum-aware directed fuzzer that integrates both dynamic taint analysis and symbolic execution. To bypass check-sum checks, it first inspects a given check field input and determines if they go across check-sum check by using taint propagation information captured during program execution. Then, it adjusts malformed inputs using symbolic execution. To reduce the time needed to generate well-formed inputs, TAINTSCOP does not consider all input bytes as symbolic values. Instead, it only replaces check-sum instances with the symbolic values on the test case and leaves the rest of bytes as a real value. TAINSCOP is evaluated over several x86 applications in both Windows and Linux, including Adobe Acrobat, Google Picasa, and ImageMagick. It uncovered 27 undocumented vulnerabilities.

Similarly, DOWSER [78] integrates static analysis, dynamic taint tracking analysis, and symbolic execution to discover buffer overflow and buffer underflow vulnerabilities located at the bottom of the program logic. Instead of exploring the whole code, DOWSER targets code areas that access the array index and performs static analysis to rank these areas based on how likely they are vulnerable. Furthermore, it applies taint tracking analysis to determine more precisely which inputs affect access to the array index. Then, it considers them as the only symbolic inputs for the subsequent symbolic execution to reduce the time complexity and to enhance the path selection. However, DOWSER assumes that there exists already a test case that helps reach the vulnerabilities places. Moreover, generating a test case is a time-consuming task.

3.2.4 Comparative Study.

We conduct qualitative comparisons based on the available information provided in each solution in terms of the approaches, implementations and evaluations as demonstrated in Table 3 and Table 4. The first and second columns of the Table 3 specify proposals and corresponding venues ordered by the date. The next seven columns present their main proposed methodologies. In next three columns, we mark which CPU architectures are supported by these approaches. The next column indicates which framework(s) have been used by these approaches. If framework have not been used,

we leave it empty. Finally, the last two columns show which tools are open source and which ones are accessible as a service to public to examine their code. Furthermore, we provide distribution of these feature over all proposed works; for instance, the majority of the solutions (48%) are employing fuzzing, while only 5% of them scan the network.

Proposals	Venue	Methodology							CPU Arch.			Reused Framework(s)	Release	
		Web Interface Check	Emulation	Symbolic Execution	Taint Analysis	Static Analysis	Fuzzing	Network Scanning	x86-64	ARM	MIPS		Open Source	Open Service
Suh et *al.* [145]	NDSS 2004				•					•				
TaintCheck [122]	NDSS 2005				•					•				
DYTAN [34]	ISTR 2007				•					•				
BinHunt [67]	ICICS 2008			•		•				•				
KLEE [23]	USENIX 2008			•						•			•	•
SAGE [70]	NDSS 2008			•						•			•	
Chess et *al.* [32]	ISTR 2008				•					•				
TaintScop [152]	SP 2010			•	•	•				•				
MAYHEM [25]	S&P 2012			•						•				
Exposé [124]	COMPSAC 2013			•		•				•		BAP		
Dowser [78]	USENIX 2013			•	•	•				•				
BLEX [55]	USENIX 2014													
Rebert et *al.* [135]	USENIX 2014						•	•		•				
SYMFUZZ [26]	SP 2015						•			•				
MACKE [125]	ASE 2016			•		•				•		KLEE		
Driller [144]	NDSS 2016			•			•			•		angr		
Angr [141]	SP 2016			•		•	•		•	•	•		•	•
Feist et *al.* [60]	SSPR 2016			•		•	•			•		GUEB		
deepFuzz [19]	DIMVA 2018			•			•			•		AFL		
CoP [106]	TSE 2017			•		•			•	•				
VUzzer [133]	NDSS 2017				•		•			•				
Godefroid et *al.* [71]	IEEE/ACM 2017						•			•				
Steelix [103]	2017					•	•			•				
SAFL [151]	ICSE 2018			•			•			•		AFL		
T-Fuzz [128]	SP 2018						•			•				
SmartSeed [107]	2018						•			•				
QSYM [159]	USENIX 2018			•			•			•				
Distribution		5.70%	13.46%	32.69/%	15.38%	19.23%	48.07%	5.07%	61.53%	30.76%	26.92%	NA	11.52%	9.61%

Table 3. A comparison of state-of-the-art dynamic and symbolic execution vulnerability detection approaches on normal binaries
(•) means that the approach presents the corresponding feature, it is empty it is empty otherwise. (–) means that the information is not provided.
Distribution presents the percentage of each category used in all proposals. The grey cells are for the sake of readability to separate different categories.

Additional qualitative and qualitative comparisons for fuzzing approaches in terms of their implementations and evaluations are introduced in Table 4. We consider symbolic execution as a white-box based fuzzer as defined in [70]. The second column categorizes the proposals based on fuzzing types including black-box, white-box, and gray-box fuzzers [108]. The third column outlines the proposals based on input types and supported formats. The forth column (Dataset) reports the number, domain and public availability of dataset used for evaluation. The fifth column provides the results reported by each proposal, such as how many vulnerabilities are discovered, how many of them are 0-day, and the number of different types of detected vulnerabilities. The sixth column specifies the platform used to run the experiments. The seventh column indicates the duration of the fuzzing campaign against the targeted Dataset reported in column four. The eight and ninth columns report the number and the name of the approaches, which are compared with the proposal.

Discussion. The key observations of this comparative study on the proposed approaches on dynamic analysis and symbolic execution are as follows: First, as can be noted from Table 3, the majority of proposed approaches have been evaluated on binary code running on traditional CPU architecture (x86 and x64). Even though many of the approaches

Proposals	Fuzzing Type						Fuzzing Input				Dataset				Reported Result							OS		Test Campaign Duration	Comparison	
	Generation_Base	Mutation_Based	Black_box	Gray_box	White_box	Coverage_based	File	Network	Web_UI	Format	Binaries	Firmware	Embedded	Available	Vulnerabilities	0-day Vulnerabilities	Type of Vulnerabilities	New-Crashes	Crashes	Bugs	New-Bugs	Linux	Windows		#of Comparison	Compared with
KLEE [23]					•	•					452									56	3			904 d		
SAGE [70]					•	•	•				7										30		•	70 h		
Koscher et al. [97]		•	•					•				2	2		25											
TaintScope [152]					•		•	•		png, jpeg, tiff, bmp, gif, pcap	8					6						•	•	80 m		
Almgren et al. [4]																										
Mulliner et al. [115]																										
Heninger et al. [80]																										
Kamel et al [90]		•	•					•		http		1			1					1				6 d		
FIE [48]					•	•						99			22		2				21			24 h		
Dowser [78]					•						6				7	2	1					•		30 m		
RPFuzzer [153]	•	•						•		Router Protocols					8	5								1440 m	3	Peach, SPIKE, Sulley
Van et al. [149]																										
Alimi et al. [3]		•	•									1	1		1		1						•	12 h		
Rebert et al. [135]		•	•			•	•			pdf, mp3, gif, pg, png	10			•	1		1		2,702	240				650 d		
SYMFUZZ [26]		•	•		•		•				8			•				110	110			•		8000 h	3	BFF,zzuf,AFL-fuzz
Lee et al. [101]																										
Firmalice [140]												3			2		1							11 h		
Driller [144]		•			•		•				126			•	6				77					24 h	2	AFL, Mayhem
Feist et al. [60]															1	1	1							20 m		AFL, radamsa
deepFuzz [19]																										
VUzzer [133]		•		•		•	•				302								1008		8	•		30 h		
Godefroid et al. [71]	•	•	•				•			pdf	1				1	1	1						•	5 d		
Steelix [103]		•		•		•	•				22					1				272	9	•		5 h	4	FUZZER, SES, Vuzzer, AFL-lafintel
SAFL [151]		•					•				10								255					24 h	2	AFL, AFLFast
T-Fuzz [128]		•		•		•	•				300			•						166	3	•	•	24 h	3	Steelix, Vuzzer, AFL
SmartSeed [107]	•						•			mp3, bmp, flv	12				23	16	9		1096			•		72 h	1	AFL
QSYM [159]		•		•			•				12									2265	13	•		3 h	2	Driller, VUzzer
IoTFuzzer [29]		•	•					•		http, https			17		15	8	3	4	4			•		408 h	2	Sulley, BED
IoTHunter [158]		•		•		•		•		snmp, ftp, ssl, bgp, smb			8		5	2						•		30 d	2	boofuzz, IoTFuzzer
WMIFuzzer [150]		•	•					•	•	http, https			7		10	6	3		3			•		23 h	2	AFL, Sulley
Firm-AFL		•			•	•		•		http, upnp		288	7		15	2						•		50 h		
Distribution	12%	65%	30%	19%	30%	35%	46%	30%	34%	NA	58%	23%	23%	15%	62%	42%	38%	8%	30%	23%	27%	46%	19%	96%	42%	NA

Table 4. A comparison of state-of-the-art fuzzing vulnerability detection evaluations.

(•) means that the approach offers the corresponding feature, otherwise it is empty.
Distribution presents the percentage of each category used in the selected proposals. The grey cells are for the sake of readability to separate different categories.

claim that their work could be adapted to support multiple CPU architectures, almost no work has been extended. Second, based on Table 4, the majority of reported bugs, vulnerabilities, or 0-day vulnerabilities are discovered by using fuzzing approaches. However, fuzzing approaches require a significant amount of time; they need days to report interesting results compared to the other approaches. Moreover, using machine learning to generate smart seed improves fuzzing marginally. For example, smartSeed [107], which utilizes the adversarial neural network to generate smart seeds, was able to discover the highest number of vulnerabilities and crashes. On the other hand, integrating fuzzing with symbolic execution or with taint analysis helps with discovering more bugs in less time. Moreover, fuzzing embedded systems through their provided network connection interface shows efficient performance by discovering more bugs and vulnerabilities with less effort compared to static analysis approaches performed after reverse engineering. Furthermore, fuzzing approaches have been employed extensively against software running on CPU architectures with modern operating systems, such as Windows or Linux, which provide build-in errors, logging, and debugging reporting services. On the contrary, embedded devices have neither similar CPU computations capabilities nor the aforementioned error reporting capabilities. This limitation makes it harder for the researchers to evaluate their approaches on these kind of devices. Third, combining two or more binary analysis approaches shows promising scalable and accuracy results such as Driller [144] and other similar approaches inspired from it. Consequently, there is potential for future research on this area. Forth, even though the second majority of vulnerabilities are discovered as a result of dynamic analysis

and symbolic execution testing, these approaches cannot be applied to large scale and complex software (such as Web browsers), or to check a large corpus of software within a reasonable time. On the other hand, in spite of their limitation, static analysis approaches could be employed to complement dynamic analysis approaches and symbolic execution; for instance, static analysis approaches can search for the similar vulnerable functions that have been discovered by fuzzing or symbolic execution, which are cross-compiled for different CPU architectures.

3.3 Discussion

In the following, we discuss the limitations of dynamic analysis approaches.

Non-trivial benchmarking. In order to benchmark the state-of-the-art approaches, representative datasets and uniform evaluation metrics are required. Most of the existing works perform their experiments on their own collected dataset and evaluate the accuracy of their system based on different metrics. There exist DARPA CGC dataset [43], LAVA [52] and LAVA-M dataset [143], however, they are not comprehensive to cover all the use cases. Therefore, for the sake of evaluation, two large-scale datasets are needed. (i) *Vulnerability dataset*, which is composed of large number of cross-architecture and cross-compiled binaries (e.g., free open-source libraries) including the vulnerable functions with the debug information. (ii) *Firmware dataset* containing a large number of publicly available firmware images and embedded system devices collected from various vendors. Therefore, existing methods could be applied on these datasets and by considering a unified evaluation metric, both the accuracy and the scalability of proposed approaches can be evaluated. Finally, a comprehensive quantitative and qualitative comparative study on the state-of-the-art approaches can be performed.

Handling multiple architectures.

Limited code coverage. Code coverage is essential for complete, and robust vulnerability discovery. Static analysis approaches would fail to discover runtime data-oriented exploits when some part of the code could be loaded dynamically, or in the case of network activities, since this information will be provided during the runtime process. On the other hand, dynamic analysis techniques might encounter issues with path exploration that would stop the process for the rest of the code. In addition, some paths may not be taken unless triggered by user interaction. Moreover, the code loaded at runtime may escape both dynamic and static analysis if the right conditions are not met (a.k.a logic bombs).

Scalability using filtering. The scalability issue of both static and dynamic analyses approaches has been somewhat addressed by filtering processes. During this process, the high-likely dissimilar or non relevant functions will be excluded from the analysis. Therefore, filtration processes minimize the search space in order to statically identify the vulnerabilities more efficiently, and also to provide a better code coverage in the case of dynamic analysis. However, the filtration process may affect the accuracy. Therefore, examining the proposed filtration processes could help better learn the pros and cons of each method, and further propose new efficient and accurate filtration techniques.

Non-trivial test case generation. To perform scalable, dynamic analysis, a suitable input sample should be available in the first place for the targeted program. Based on the literature review, test cases have been generated either manually by experts, using symbolic execution, or by a trained machine learning model. Manual test case generation by an expert is a long-time process, not scalable, and costs a lot. On the other hand, symbolic execution is also not scalable, cannot provide high coverage for complex software, and it is computationally expensive. Furthermore, proposed solutions that combine symbolic execution with fuzzing to solve this issue are not suitable yet for complex software (e.g., web browsers, and mobile applications). Recently, machine learning approaches have been proposed to generate input

samples from provided training samples automatically; however, these solutions have been trained and tested over simple applications, such as simple PDF format or media player files. On the other hand, surveyed dynamic analysis approaches have helped with discovering variant types of vulnerabilities. Therefore, one potential open research area is integrating an automatic test case generation using machine learning with fuzzing or a hybrid approach. Moreover, Lie et al. [102] recommend that a new fuzzing approach should also take into account understanding vulnerabilities types as well as characteristics in addition to the execution traces, as feedback during a fuzzing campaign.

Fuzzing limitations for embedded systems. Different fuzzing techniques have been proposed and evaluated on traditional computing environment, such as PCs and servers, however, not all of them are applicable for embedded system firmware [114]. In addition to limited resources of embedded system devices (e.g., CPU and RAM), the majority of the embedded systems operating system do not implement memory management module. Consequently, in the case of a memory crash, no report is generated. Therefore, building an emulator to report the memory corruption and to provide feedback on the implemented fuzzing approach will open a chance to examine how variant proposed fuzzing approaches are effective against embedded systems. Although emulators (e.g., QEMU) have a high impact on the performance of the emulated platform, they do not emulate all other connected peripherals easily. As such, they may fail in creating a rich environment to test potential vulnerabilities effectively. Therefore, it is highly desirable to have an emulator with common emulated components that can be added to the emulation or removed to replicate the most likely environment in which the firmware is to be deployed.

Emulations limitations for embedded systems. Proposed emulation solutions for embedded systems are still in their infancy period. Building partial or full emulation for embedded systems is a daunting and complex process, since embedded system manufactures do not publish their product documentation and the setup environment, which is used to test and protect their intellectual property. Even though there have been prominent works in this field, they suffer from the following limitations: supporting only limited CPU architectures (e.g., ARM and MIPS), not supporting generic interrupt handling, and supporting limited number of peripherals modelling. Tackling these limitations still remains as an open area of research.

Symbolic execution inefficiency. Proposed symbolic execution techniques have shown their usefulness in detecting more vulnerabilities either when they are applied as a stand-alone approach, or when they are integrated with other binary analysis techniques. However, reducing the time complexity of the constraint solvers is still a bottleneck.

4 STATIC ANALYSIS

In this section, we aim at reviewing static approaches that are applied to embedded system firmware images. To the best of our knowledge, there is no static analysis work that identifies the vulnerabilities in embedded systems firmware images. There exist a limited number of works that employ static analysis to find vulnerabilities in normal binaries. For instance, few static analysis approaches [61, 134, 142] are proposed to locate Use-After-Free vulnerabilities [111], and different buffer overflow vulnerabilities. The code-similarity problem focuses on statically analyzing two pieces of binary code (e.g., functions) in order to measure their similarities. A function is deemed as a potential vulnerable function, if there is a match between that function and an already analyzed vulnerable function in the repository. In order to overcome the scalabiliy issue (C9), some of the existing works propose filtering prior to the comparison. The objective of the filtering process is to exclude high likely dissimilar functions from the analysis to prune the search space while maintaining the accuracy.

Various static-based solutions based on code-similarity comparison on both normal binary code and firmware images have been proposed in the literature. In this section, we review those solutions that are applied to firmware images. We

first introduce different features used in static solutions, then briefly explain existing code similarity approaches for vulnerability detection on firmware images (based on the taxonomy shown in Figure 2). Finally, we compare those approaches in terms of methodologies, implementations and evaluations, followed thereafter by a related discussion.

4.1 Taxonomy of Features

In order to analyze program binaries, existing works extract a wide range of features from different binary code representations or intermediate representations (IR), a processor-neutral form, which represents the operational semantics of the binary code with an intermediary level of abstraction. Then, these features (or the combination of them) are employed to represent the semantics of a function or a program binary and ultimately help analyzing a price of code. We classify the features into four categories. Figure 3 shows the proposed taxonomy of features as well as the corresponding approaches that utilize these features.

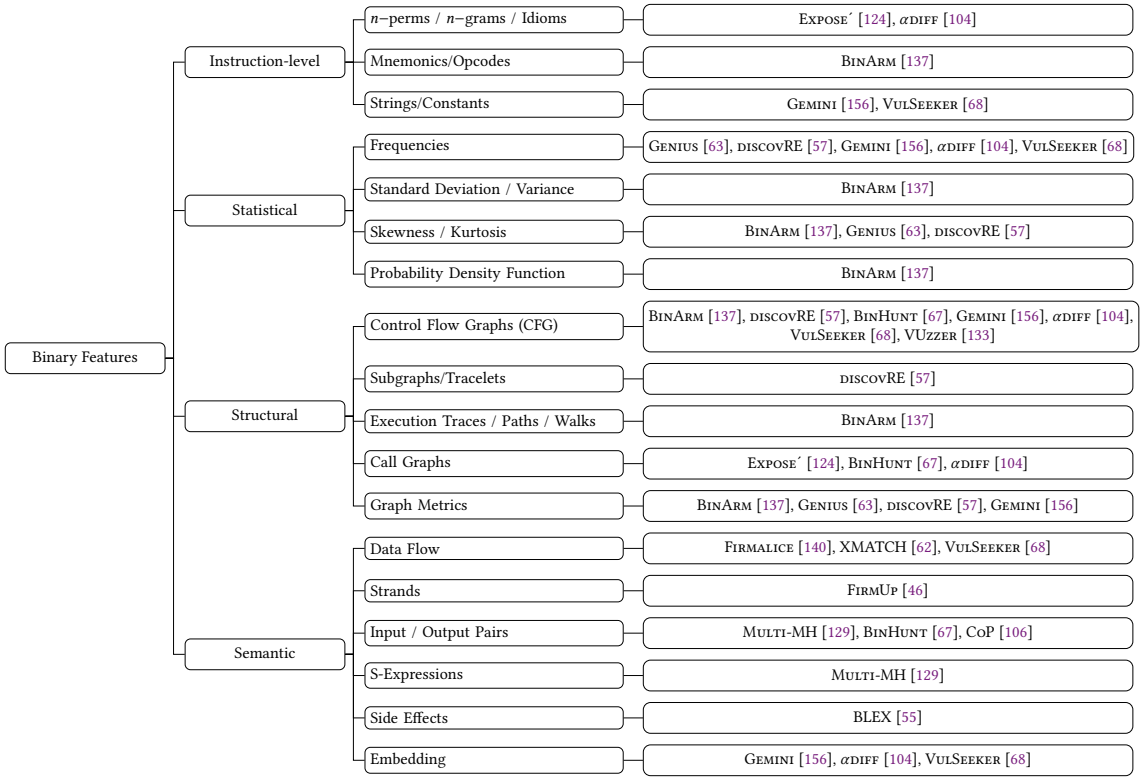

Fig. 3. The proposed taxonomy of the features and corresponding existing approaches

4.1.1 Instruction-level Features. Instruction-level features can be extracted directly from a given binary code. For instance, *n-grams* [117] are n sequences of tokens (e.g., bytes or instructions) in a program binary. One drawback of the *n-grams* is that they are sensitive to the order of instructions, since some instruction could be reordered while preserving the semantics. To solve this issue, *n-perms* [95] approach is proposed, which involve n sequences of tokens with any order. *Idioms* are composed of short sequences of instructions with wildcards, where the values of immediate operands

are abstracted away [136]. A *mnemonic* of an instruction represents the operations that need to be executed, whilst typically an *opcode* is the hexadecimal encoding of the instruction [96]. *Strings* and *constants* are other instruction-level features that can be captured easily from the instruction sets. The motivation of using constants as a feature is that usually constants remain unchanged regardless of the compilers and optimization settings.

4.1.2 Statistical Features. Statistical features represent the semantic information of a binary code, for instance, cryptography functions use more arithmetic and logical instructions compared to a function which writes some information into a file. For this purpose, the statistics of different features, such as *frequencies*, are used in the literature. As an example, instructions grouping [99] (e.g., number of arithmetic instruction) is utilized to get more information about the functionality of a function. *Opcode distributions* [16] are used to detect metamorphic malware. Additionally, some statistical distributions such as *skewness* and *kurtosis* [118], and *probability density functions* are computed to get more information about a function and the distribution of its instructions [45, 137, 138].

4.1.3 Structural Features. Structural features represent the semantic information as well as the structural property of a code. *Control flow graphs* (CFGs) [64] are the most frequent features used in the literature to represent the control flow of the execution. A subset of CFGs called *tracelets* [47], *partial traces* [27] or *subgraphs* [99], which are defined as the short and partial traces of an execution, capture the semantic of execution sequences. Similarly, *execution traces/paths/walks* [5, 27, 137] represent the execution traces, while considering basic block semantics. *Call graphs* [64] are extracted at the program level to get more information about both the relation between callers and callees as well as the program logic. Additionally, some graph metrics [74], such as *graph energy* and *betweenness centrality* (node centrality) [121], are used to extract more information about the topology of the graph.

4.1.4 Semantic Features. This category includes features that convey the code semantics to a larger degree. *Execution flow graphs* (EFG) [132] preserves the data and control dependencies between the instructions in a CFG and therefore, represents the internal structure of a function. *Data dependence* and *program dependence graphs* (PDGs) [112, 139] are used to reason about the control and data flow, where memory and register values are required to be extracted. Data flow analysis combined with path slicing [85] and value-set analysis [11] are employed to construct *conditional formulas* [62], which describe when a given action will take place under which condition and could capture incorrect data dependencies and condition checks. *Strands* [44] are the set of instructions resulting from backward slicing [154] at basic block level. *Input/Output pairs* [27, 67, 89, 105, 106, 129] are obtained by the assignment formulas for each basic block, where the effects of input variables on the output variables are monitored in order to capture their semantics. *S-Expressions* [130] are a tree-like data structures composed of equations, which capture the effect of basic blocks on the program state. The equations are obtained from the basic block instructions, where both left-hand and right-hand sides contain arbitrary computations. *Side effects* [55] are composed of a set of features, such as values written to (read from) program heap, system calls, etc., that are collected during the program execution and preserve function semantics. *Embeddings* [156, 166] are high-dimension numerical vectors obtained from a function or a fragment of it (e.g., CFG or basic blocks), which preserve and convey the meaning of the functions.

4.2 Code Similarity Detection

In this subsection, we review code similarity detection approaches that identify vulnerabilities in firmware images. We categorize the existing works into three groups of *graph-based*, *data flow-based* and *distance-based* explained as follows.

4.2.1 *Graph-based Approaches.*

Graph-based approaches perform the analysis based on graph representation of a piece of code, such as control flow graph (CFG), subgraphs, tracelets and call graphs, each of which convey specific information.

To perform vulnerable function detection in cross-compiled cross-architectures firmware images, DISCOVRE [57] is proposed. It extends the maximum common subgraph (MCS) [109] distance to additionally consider the similarity between basic blocks, based on some features, such as topological order in the function, strings, and constants. Since MCS execution time will grow exponentially, the authors employ a numerical filter based on a set of features (e.g., number of instructions, number of parameters, local variable sizes, and number of incoming/outgoing edges) and the KNN algorithm [40], and further terminate the algorithm after a certain number of iterations. DISCOVRE is tested on the firmware images of the DD-WRT router, NetGear ReadyNAS, and Android ROM image. Nevertheless, according to the performed evaluation in [63], the utilized pre-filtering causes notable deduction in accuracy. Therefore, inspired by DISCOVRE, GENIUS [63] utilizes both statistical and structural features that are consistent among multiple CPU architectures, and labels each basic block in a CFG with the set of attributes to construct the attributed control flow graph (ACFG). To perform an efficient searching process, the ACFGs are converted into codebooks using spectral clustering [123] and further encoded [8] using a high-level embedding and locality sensitive hashing (LSH). However, the authors state that creating the codebook is expensive. The bug search is performed on 8,126 firmware images from 26 different vendors, such as ATT, Verizon, Linksys, D-Link, Seiki, Polycom, and TRENDne. This includes different products, such as IP cameras, routers, and access points. Similar to DISCOVRE, GENIUS is evaluated on DD-WRT router, and NetGear ReadyNAS firmware images.

A multi-stage detection engine, called BINARM [137], based on a coarse to fine-grain detection approach is proposed to efficiently identify vulnerable functions in the intelligent electronic devices in the smart grid. In the first stage, the candidate functions which have a certain Euclidean distance (based on the skewness, kurtosis and graph_energy) from a given function are discarded. The second stage, drops the candidate functions that have different execution paths by leveraging the probability density function and TLSH [126]. Finally, fuzzy graph matching using weighted Jaccard similarity and Hungarian algorithm is employed to identify the vulnerable functions. BINARM is evaluated on 5,756 ARM-based firmware images including the NetGear ReadyNAS firmware image.

4.2.2 *Data Flow-based Approaches.*

Data flow-based approaches typically observe the flow of data by analyzing the memory reads and writes, input/output pairs, variable locations, etc. However, most of the existing data flow-based solutions cannot be apply at large scale. For instance, MULTI-MH [129] derives bug signatures in the form of subgraphs from both source code and program binaries to identify vulnerable functions on multiple CPU architectures. First, assembly instructions are lifted into RISC-like expressions using VEX-IR [120] to obtain assignment formulas, and then the assignment formulas are simplified to S-Expressions by leveraging $Z3$ theorem prover [49]. Second, input/output behavior of assignment formulas are sampled by using random concrete input values to capture the basic block semantics. Afterwards, MinHash [89] is used to reduce the complexity of similarity measurement amongst two basic blocks. Finally, in order to match the entire signature with a given target function, a greedy but locally-optimal graph matching algorithm called Best-Hit-Broadening (BHB) is proposed. BHB algorithm first performs basic block matching and further explores the immediate neighborhood nodes using Hungarian method [65] (with no backtracking) to identify additional optimal matches. However, the proposed approach is not practical at large scale, since k-MinHash degrades the performance quite significantly [129]. MULTI-MH is examined on the DD-WRT, NetGear, SerComm, and MikroTik firmware images.

Another approach called FirmUp [46] identifies vulnerable functions in firmware images by considering the relationships between the functions. First the functions are decomposed into basic blocks, and then slicing is applied on the basic blocks to obtain the strands. Further, compiler optimizer and normalizer are utilized to transfer the semantically equivalent strands to a canonical form (syntactic form). The more the functions share the same strands, the more they are similar. To improve the accuracy, a back-and-forth games algorithm [56], called Ehrenfeucht-Fraïssé, is leveraged to perform the matching for the neighboring functions and therefore to extend a more appropriate partial matching. FirmUp is tested on about 2000 firmware images from various device vendors, including NetGear, D-Link and ASUS.

A semnatic-based approach called XMATCH [62] searches for vulnerable functions in a cross-architecture cross-platform environment. XMATCH builds *conditional formulas* from binary code functions, where the instructions are lifted to IR using McSema [51]. The *conditional formulas* are robust against CFG structure and CPU architecture variations. XMATCH utilizes both data dependencies and condition checks, and can detect erroneous data dependencies and both absent or incorrect condition checks. It is tested on the firmware image of the Linux-based DD-WRT router.

4.2.3 *Distance-based Approaches.*

The distance-based approaches extract different sets of features for a function, and then various similarity metrics are applied on the selected features in order to find the matching pairs. A cross-architecture binary code similarity approach called Gemini [156] is proposed based on a neural network model. It first extracts the control flow graphs attributed with manually selected features called attributed control flow graphs (ACFG), and then employs `structure2vec` [42] combined with Siamese architecture [20] in order to generate the graph embeddings of two similar functions close to each other. Introducing embedding with deep learning by Gemini highly improves binary function fingerprinting over multi-platform CPU architectures. However, embedding generated by Gemini relies on mainly statistical features without considering the relationships between them and the instruction sets represented by these features. The reported vulnerability identification accuracy of about 82% reflects the limitation of such feature choices to be applied to vulnerability detection problem. Similar to Genius, 8,128 firmware images from 26 vendors with different products, such as IP cameras, routers, and access points are indexed in their repository.

Inspired by Genius, VulSeeker [68]extracts the labeled semantic flow graph (LSFG), which combines CFG with DFG, and then propose a semantics-aware deep neural network model in order to generate the function embeddings. Finally, the cosine similarity is used to measure the similarity between two functions. VulSeeker is examined on 4,643 cross-architecture firmware images.

An approach called, αdiff [104], extracts function code (raw bytes), function calls and function's imported functions to perform cross-version binary code similarity detection. The convolutional neural network (CNN) and a Siamese network are used to convert the function code into embeddings. Three distances, including inter-function distance, intra-function distance and inter-module distance are calculated. Finally, the overall distance from a given function to the vulnerable functions in the repository is measured. Similar to most of the previously mentioned works, αdiff is evaluated on DD-WRT, and NetGear ReadyNAS firmware images.

4.3 Comparative Study

A comprehensive evaluation of the existing works is not feasible, mainly due to the absence of their dataset and the exact firmware images. However, we conduct a qualitative comparison based on the information provided by each solution in terms of the approaches, implementations and evaluations. Table 5 and Table 6 summarize the findings of this study. The first and second columns of Table 5 specify the proposals and the corresponding venues ordered by the

date. The next four columns present the features used by each proposal, based on our proposed features taxonomy presented in Figure 3. The next three columns indicate the types of analysis based on our taxonomy presented in Figure 2. In the next column, we provide the corresponding main proposed methodologies. The next two columns provide the disassembler as well as the use of intermediate representation. Afterwards, the "Mapping Results" column indicates the presence of in-depth analysis. It is marked when a work provides the matching results (e.g., corresponding matched instruction sets or basic blocks) in addition to a final similarity score. Finally, the last two columns show which tools are open source and which ones are accessible to the public. Furthermore, we provide the distribution of different features (e.g., types of analysis) among different surveyed works, which are shown in the last row. For instance, 56% of the exiting solutions rely on semantic features while only 22% extract instruction-level features.

Proposal	Venue	Feature(s)				Analysis			Methodology	Disassembler	IR	Mapping Results	Release	
		Instruction-level	Semantic	Structural	Statistical	Graph-based	Data Flow-based	Distance-based					Open Source	Open Service
Multi-MH [129]	S&P'15		●	●			●		BHB, MinHash	IDA	VEX [120]			
discovRE [57]	NDSS'16			●	●		●		MCS, JD	IDA				
Genius [63]	CCS'16			●	●	●			LSH, JD	IDA				
FirmUp [46]	ASPLOS'18		●				●		DFA, SR	IDA	VEX, LLVM			
Gemini [156]	CCS'17				●			●	DNN	IDA			●	
XMATCH [62]	ASIACCS'17		●	●			●		DFA,GED	IDA	McSema [51]	●		
BinArm [137]	DIMVA'18	●	●	●	●	●			WJD, ED, GM	IDA		●		
VulSeeker [68]	ASE'18		●	●				●	DNN	IDA			●	
αdiff [104]	ASE'18	●		●				●	DNN	IDA				
Distribution	NA	22%	56%	78%	44%	22%	44%	33%	NA	NA	33%	22%	22%	0%

Table 5. A comparison of state-of-the-art static-based code similarity detection approaches

(●) means the approach provides the corresponding feature, it is empty otherwise. (BHB) Best-Hit-Broadening, (DFA) Data Flow Analysis, (DNN) Deep Neural Network, (GED) Graph Edit Distance, (JD) Jaccard Distance, (LSH) Locality Sensitive Hashing, (MCS) Maximum Common Subgraph, (SR) Statistical Reasoning, (WJD) Weighted Jaccard Distance. Distribution presents the percentage of each feature category that is used in all proposal. The grey cells are for the sake of readability to separate different categories.

We further conduct comparative study on the implementation and evaluation of existing works as listed in Table 6. The first column lists the proposals. The second column presents used programming languages in each proposal. The next two columns indicate the dataset (normal binary, and firmware) used for the experiments. The next four columns mark employed compilers utilized to prepare the ground truth. In next three columns, we mark which CPU architectures are supported by these approaches. In the next two columns, the operating systems is marked. Afterwards, the "Detection" criteria provides the type of vulnerable function detection. It is marked when a work identifies known vulnerable functions, or unknown vulnerabilities. The next three columns show works that use normalization and provide accuracy and performance results. Next column indicates the number of works which have been compared with the current work. The last columns marks whether the work is using any filtering process. We also provide different distributions, e.g., the ratio of the works that support different CPU architectures. For instance, 90% of the exiting solutions support GCC compiler while only 11% support ICC and VS compilers.

4.4 Discussion

The key observations of the performed comparative study are as follows: First, there exist several features in the literature which are shown to significantly improve the efficiency and accuracy of the vulnerability detection solutions.

Proposal	Programing Lan.	Dataset		Compiler(s)				CPU Arch.			OS		Detection		Normalization	Accuracy	Performance	Comparison	Filtering
		Normal Binary	Firmware	VS	GCC	ICC	Clang	x86-64	ARM	MIPS	Window	Linux	Known Vul.	Unknown Vul.					
Multi-MH [129]	C++	60	4		●		●	●	●	●	●	●	●			●	●	0	
discovRE [57]	-	2, 280	2	●	●	●	●	●	●	●	●	●	●			●	●	2	●
Genius [63]	Python	17, 626	8, 128		●		●	●	●	●		●	●			●	●	3	●
FirmUp [46]	-	200, 000	2, 000	-	-	-	-	●	●	●		●	●		●	●	●	2	
Gemini [156]	Python	51, 314	8, 128		●			●	●	●	●	●	●			●	●	2	
XMATCH [62]	-	72	1		●		●	●		●	●	●	●			●	●	4	
BinArm [137]	C++, Python	-	5, 756		●				●			●	●		●	●	●	5	●
VulSeeker [68]	Python	-	4, 643		●			●	●	●		●	●			●	●	1	
αdiff [104]	-	67, 427	2		●		●	●	●	●		●	●			●		6	
Distribution	NA	100%	100%	11%	90%	11%	56%	90%	90%	90%	44%	100%	100%	0%	33%	100%	90%	100%	33%

Table 6. A comparison of state-of-the-art static-based code similarity detection implementations and evaluations
(●) means that the approach provides the corresponding feature, it is empty otherwise. (–) means that the information is not provided.
Distribution presents the percentage of each feature category that is used in all proposals. The grey cells are just for the sake of readability to separate different categories.

As can be observed, semantic and structural features are the most frequently used features. Second, there is no single solution to identify vulnerable functions. Among which, data flow-based approaches demonstrate the best practice to be chosen for the vulnerability detection in firmware images. More recently, distance-based approaches which employ DNN and NLP show the best results for cross-architecture vulnerability detection. Third, the filtering process is a promising solution to overcome the scalability issue. However, these filtering approaches should be carefully designed and thoroughly evaluated to assure the accuracy. Forth, none of the code similarity solutions can identify unknown vulnerabilities. Finally, even though MinHashing and LSH are employed for function matching, existing works under this category are not practical at large scale due to their time complexity for functions with large and complex control flow graphs. To conclude, most of the recent static-based code similarity solutions employ data flow-based approaches over x86, ARM and MIPS architectures for Linux-based firmware images compiled with GCC compiler. Moreover, this comparison demonstrates the trend of analysis which is moving towards DNN and NLP techniques on Linux platform to overcome cross-architecture problem and the scalability issue of online searching. As seen, the existing approaches can overcome some of the challenges, such as compiler effects (C2) and hardware architecture (C5). However, still some of them remain unresolved. In the following, we discuss the limitations of static-based solutions.

Detecting unknown vulnerabilities. Most of the static solutions, define a pattern/signature for a function, and then perform function matching. Therefore, already known vulnerable functions are stored in the repository and by identifying any match with them, the vulnerable functions are discovered. However, there might be some functions with unknown vulnerabilities, in which could not be identified in this manner (C6). Unknown vulnerabilities might be identified by employing data flow and dynamic analysis.

Detecting run-time vulnerabilities. Static approaches fail to detect vulnerabilities that are exploited during the execution time (C6). For instance, the run-time data-oriented exploits cannot be detected due to the lack of execution semantics checking [31], or any other vulnerabilities that need network communication.

Identifying inline functions. Function inlining (C4) may introduce additional complexity to the vulnerable function detection problem, since it requires to fingerprint a function with partial code from another function. Static approaches

generally fail to identify inline functions. However, data flow analysis and symbolic execution could be employed as potential solutions to this problem. Systematically addressing this problem is still an open challenge.

Scalability using filtering. The scalability issue (C9) of both static and dynamic analyses approaches has been somewhat addressed by filtering processes. During this process, the high-likely dissimilar or non relevant functions will be excluded from the analysis. Therefore, filtration processes minimize the search space in order to statically identify the vulnerabilities more efficiently, and also to provide a better code coverage in the case of dynamic analysis. However, the filtration process may affect the accuracy. Therefore, examining the proposed filtration processes could help better learn the pros and cons of each method, and further propose new efficient and accurate filtration techniques.

Lack of semantic insights and replaying vulnerabilities. Static approaches provide a list of potential vulnerable functions with relatively high false positives rates (C6). Therefore, manual effort is required in order to verify the obtained vulnerability results. These techniques do not provide any information on how to trigger the discovered vulnerabilities for further investigation and to replay the attacks (C7). Therefore, other approaches (e.g., symbolic execution) could be employed to produce repayable inputs in order to validate the bugs and further provide semantic insight on the reason of the execution and the corresponding part of the code. On the other hand, static approaches could be employed to overcome the scalability issue of pure dynamic analysis and symbolic execution techniques.

Generalizing vulnerability signatures. Most of the existing approaches provide a specific pattern in different representations and semantic levels for each vulnerable function, and then employ a matching technique or a similarity measurement to identify it. Providing a general signature for each vulnerability (e.g., buffer overflow) rather than matching with the functions that already have a specific vulnerability is one of the future directions.

5 LESSONS LEARNED AND FUTURE DIRECTIONS

According to the performed survey, and considering the fact that each technique has its own advantages and disadvantages, we first list the limitations which are common between static and dynamic analyses approaches, and then provide the limitation which are specific for each approach. Finally, we discuss some proposed future work directions.

Based on our analysis, the general learn lesson is that, variant embedded system devices, which are utilized at variant sectors, have many vulnerabilities and bugs. Due to their exponential increase of their usage, they become a priceless and easy target for the attackers.

In the following, we outline specific lessons learned to help practitioners to test embedded device for known and unknown vulnerabilities.

(1) practitioners could first start by inspecting the target device contains well-known vulnerable functions by utilizing similarity approaches in static analysis . Embedded system firmware developer generally utilize software libraries or package without checking if they are vulnerability free. Using one of the state-of-the-art similarity approaches, explained in section 4.2, to locate well-known vulnerable functions will save the practitioners time and prevent potential risk by isolating them at early stage.

(2) practitioners could secondly re-host targeted device firmware outside its environment for scalable testing. if embedded system firmware could be extracted successfully, it would be better to extensively test in scalable manner outside its environment as outlined in section 3.1.1. It is more scalable to utilize one of the full-emulation approaches, but if the firmware requires frequent accesses to its peripherals, in this case the practitioners could resort to use partial-emulation or virtualization so that peripheral request will be forwarded to the hardware. Afterwards, the

targeted embedded system firmware could be tested by using one of the advanced binary analysis techniques, such as fuzzing or symbolic execution.

(3) Test embedded device over its communication interface . If embedded system firmware emulation is difficult or impossible but the device has a communication interface, then the practitioners could utilize one of the approaches mentioned in section 3.1.3 to test the targeted device over one or several of its provided communication protocol. Practitioners in this case should take into account if the used protocol is state or stateless to consider the right approach during the fuzzing campaign.

5.1 Future Research

In the following, we outline further research to be established in firmware re-hosting, fuzzing and hybrid approaches.

Automatic firmware re-hosting. Currently available re-hosting techniques (Section 3.1.1), helped with enable advanced binary analysis approaches, such as gray-box fuzzer or symbolic execution into embedded system testing world. However, proposed emulation solutions for embedded devices are still in their infancy period. Building partial or full emulation for embedded systems is a daunting and complicated process. Moreover, the re-hosting process still requires manual adjustment to make it work for variant CPU architectures. As reported in [116], Eric et al. [76] approach is an initial good work in this direction, and further research in an automatic re-host generation will open doors for scalable binary firmware analysis.

Embedded system fuzzing. Although re-hosting techniques help with introducing advanced fuzzing techniques into embedded device testing, some advanced fuzzing techniques are not proposed yet into embedded device testing world, such as test case generation, and instrumentation. These techniques are well-known for widen test coverage and help with discovering vulnerable paths.

Introducing hybrid approaches into embedded system testing. Although variant binary analysis techniques have been introduced into embedded devices testing world, a hybrid approach that integrates two or more approaches has not been utilized yet to check embedded devices for potential vulnerabilities or bugs. For example, fuzzing and symbolic execution have been combined with other methods such as taint analysis or static analysis. They establish better results from when they are utilized individually. Therefore, integrating two or more approaches together for testing embedded devices still an open area of research.

5.2 Common Limitations

In the following, we discuss the common limitations that we identified in both static and dynamic analysis approaches.

6 CONCLUSION

In this paper, we surveyed various types of approaches and methods proposed to identify vulnerabilities in binary code and embedded devices firmware. In addition, we devised a taxonomy of different types of utilized features, application domains and analysis techniques followed by a qualitative comparison, which demonstrated the trend of binary analysis techniques, its importance and the existing limitations. Finally, we discussed the lessons learned and proposed several areas of future research in this domain. As future work, we intend to conduct a quantitative comparison among the existing methods and investigate more on the parameters, which strongly couple with the efficiency, accuracy and scalability of a vulnerability detection solution for embedded devices.

REFERENCES

[1] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. 2018. A Taxonomy of Software Integrity Protection Techniques. (2018).

[2] Chengwei Ai, Weiyu Dong, and Zicong Gao. 2020. A Novel Concolic Execution Approach on Embedded Device. In *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*. 47–52.

[3] Vincent Alimi, Sylvain Vernois, and Christophe Rosenberger. 2014. Analysis of embedded applications by evolutionary fuzzing. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 551–557.

[4] Magnus Almgren, Davide Balzarotti, Jan Stijohann, and Emmanuele Zambon. 2014. D5.3 report on automated vulnerability discovery techniques. *CRISALIS EU Project* (2014).

[5] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2018. FOSSIL: A resilient and efficient system for identifying FOSS functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)* 21, 2 (2018), 8.

[6] Amazon. 2018. Amazon elastic compute cloud. (2018). https://aws.amazon.com/ec2/

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *USENIX Security Symposium*.

[8] Relja Arandjelovic and Andrew Zisserman. 2013. All about VLAD. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*.

[9] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein gan. *arXiv preprint arXiv:1701.07875* (2017).

[10] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 12–22.

[11] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. Springer, 5–23.

[12] Gogul Balakrishnan, Thomas Reps, David Melski, and Tim Teitelbaum. 2008. WYSINWYX: What you see is not what you execute. In *Verified software: theories, tools, experiments*. Springer, 202–213.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono DâĂŹelia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[14] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to recognize functions in binary code. USENIX.

[15] Clark Barrett, Daniel Kroening, and Thomas Melham. 2014. Problem solving for the 21st century: Efficient solver for satisfiability modulo theories. (2014).

[16] Daniel Bilar. 2007. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics* 1, 2 (2007), 156–168.

[17] Hristo Bojinov, Elie Bursztein, Eric Lovett, and Dan Boneh. 2009. Embedded management interfaces: Emerging massive insecurity. *BlackHat USA* 1, 8 (2009), 14.

[18] Boofuzz. 2019. 2Binwalk: firmware analysis tool. (2019). https://boofuzz.readthedocs.io/en/lates

[19] Konstantin Böttinger and Claudia Eckert. 2016. Deepfuzz: triggering vulnerabilities deeply hidden in binaries. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 25–34.

[20] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*. 737–744.

[21] Teresa Nicole Brooks. 2018. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. In *Science and Information Conference*. Springer, 1083–1102.

[22] David Brumley, Ivan Jager, Edward J Schwartz, and Spencer Whitman. 2013. The BAP handbook. (2013).

[23] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, Vol. 8. 209–224.

[24] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[25] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 380–394.

[26] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 725–741.

[27] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.

[28] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*.

[29] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*.

[30] Ting Chen, Xiao-Song Zhang, Xiao-Li Ji, Cong Zhu, Yang Bai, and Yue Wu. 2014. Test generation for embedded executables via concolic execution in a real environment. *IEEE Transactions on Reliability* 64, 1 (2014), 284–296.

[31] Long Cheng, Ke Tian, and Danfeng Daphne Yao. 2017. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 315–326.

[32] Brian Chess and Jacob West. 2008. Dynamic taint propagation: Finding vulnerabilities without attacking. *Information Security Technical Report* 13, 1 (2008), 33–39.

[33] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.

[34] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 196–206.

[35] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[36] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-wide security testing of real-world embedded systems software. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 309–326.

[37] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 437–448.

[38] Ang Cui, Michael Costello, and Salvatore J Stolfo. 2013. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *NDSS*.

[39] Ang Cui and Salvatore J Stolfo. 2010. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*. ACM, 97–106.

[40] Padraig Cunningham and Sarah Jane Delany. 2007. k-Nearest neighbour classifiers. *Multiple Classifier Systems* 34, 8 (2007), 1–17.

[41] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *NDSS*. Citeseer.

[42] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*. 2702–2711.

[43] DARPA. 2018. Cyber Grand Challenge. (2018). http://cybergrandchallenge.co

[44] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.

[45] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of Binaries through re-Optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 79–94.

[46] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 392–404.

[47] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *ACM SIGPLAN Notices* 49, 6 (2014), 349–360.

[48] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security Symposium*. 463–478.

[49] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[50] die.net. 2018. Determine file type. (2018). https://linux.die.net/man/1/fil

[51] Artem Dinaburg and Andrew Ruef. 2014. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*.

[52] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 110–121.

[53] Dominic Rath. 2018. OpenOCD. (2018). http://openocd.or

[54] Pavel Dovgalyuk. 2012. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In *CSMR*. 553–556.

[55] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX.

[56] Andrzej Ehrenfeucht. 1961. An application of games to the completeness problem for formalized theories. *Fund. Math* 49, 129-141 (1961), 13.

[57] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.

[58] ESET. 2018. Vulnerabilities reached a historic peak in 2017. (2018). https://bit.ly/2Mgk4x

[59] F-Secure. 2015. Vulnerabilities in Foscam IP cameras enable root and remote control,. (2015). https://bit.ly/2PONhRW

[60] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. 2016. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. ACM, 2.

[61] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques* 10, 3 (2014), 211–217.

[62] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 346–359.

[63] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 480–491.

[64] Halvar Flake. 2004. Structural comparison of executable objects. In *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics*. Citeseer, 161–174.

[65] András Frank. 2005. On Kuhn's Hungarian methodâĂŤa tribute from Hungary. *Naval Research Logistics (NRL)* 52, 1 (2005), 2–5.

[66] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification*. Springer, 519–531.
[67] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.
[68] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 896–899.
[69] François Gauthier, Thierry Lavoie, and Ettore Merlo. 2013. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 209–218.
[70] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing. In *NDSS*, Vol. 8. 151–166.
[71] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 50–59.
[72] Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).
[73] Andy Greenberg. 2017. The Reaper IoT Botnet Has Already Infected a Million Networks. (2017). https://bit.ly/2SiYZpJ
[74] C Griffin. 2012. Graph Theory: Penn State Math 485 Lecture Notes. (2012). http://www.personal.psu.edu/cxg286/Math485.pd
[75] GUEB. 2018. Static analyzer detecting use-after-free on binary. (2018). https://github.com/montyly/gueb
[76] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2019. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 135–150.
[77] H. Craig. 2019. 2Binwalk: firmware analysis tool. (2019). https://github.com/ReFirmLabs/binwalk
[78] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*. 49–64.
[79] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 63–72.
[80] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium*, Vol. 8. 1.
[81] Allen D Householder and Jonathan M Foote. 2012. *Probability-based parameter selection for black-box fuzz testing.* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.
[82] http://aluigi.altervista.org/mytoolz.htm. 2018. Signsrch signature identification too. (2018). http://aluigi.altervista.org/mytoolz.ht
[83] IT Governance Blog. 2018. 6 reasons why software is becoming more vulnerable to cyber attacks. (2018). https://bit.ly/2tJq7n
[84] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire OS distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 48–62.
[85] Ranjit Jhala and Rupak Majumdar. 2005. Path slicing. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 38–47.
[86] Tiantian Ji, Yue Wu, Chang Wang, Xi Zhang, and Zhongru Wang. 2018. The Coming Era of AlphaHacking? A Survey of Automatic Software Vulnerability Detection, Exploitation and Patching Techniques. In *Third International Conference on Data Science in Cyberspace (DSC)*. IEEE, 53–60.
[87] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.
[88] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. 2016. Survey on Software Vulnerability Analysis Method Based on Machine Learning. In *Data Science in Cyberspace (DSC), IEEE International Conference on*. IEEE, 642–647.
[89] Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. 2012. Binary function clustering using semantic hashes. In *Machine Learning and Applications (ICMLA), 2012 11th International Conference on*, Vol. 1. IEEE, 386–391.
[90] Nassima Kamel and Jean-Louis Lanet. 2013. Analysis of HTTP protocol implementation in smart card embedded web server. *International Journal of Information and Network Security* 2, 5 (2013), 417.
[91] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
[92] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*.
[93] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 329–340.
[94] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*. ACM, 46–53.
[95] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. 2005. Malware phylogeny generation using permutations of code. *Journal in Computer Virology* 1, 1-2 (2005), 13–23.
[96] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 329–338.
[97] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium*

on. IEEE, 447–462.

[98] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. {SURROGATES}: Enabling near-real-time dynamic analyses of embedded systems. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*.

[99] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Polymorphic worm detection using structural information of executables. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 207–226.

[100] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, Vol. 13. 18–18.

[101] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. 2015. Fuzzing can packets into automobiles. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 817–821.

[102] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6.

[103] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 627–637.

[104] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 667–678.

[105] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 389–400.

[106] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177.

[107] Chenyang Lv, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, Pan Zhou, and Jing Chen. 2018. SmartSeed: Smart Seed Generation for Efficient Fuzzing. *arXiv preprint arXiv:1807.02606* (2018).

[108] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[109] James J McGregor. 1982. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience* 12, 1 (1982).

[110] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.

[111] Mitre. 2018. CWE-416: Use after free. (2018). https://cwe.mitre.org/data/definitions/416.html

[112] Steven S Muchnick et al. 1997. *Advanced compiler design implementation*. Morgan Kaufmann.

[113] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, Vol. 18. 1–11.

[114] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings 2018 Network and Distributed System Security Symposium, San Diego, CA*.

[115] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. 2011. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *USENIX Security Symposium*. 99.

[116] Marius Munch. 2019. Dynamic Binary Firmware Analysis: Challenges & Solutions. (2019).

[117] Ginger Myles and Christian Collberg. 2005. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*. ACM, 314–318.

[118] Mary Natrella. 2010. NIST/SEMATECH e-handbook of statistical methods. (2010). http://www.itl.nist.gov/div898/handbook/

[119] Jose Nazario. 2007. BlackEnergy DDoS Bot Analysis. (2007).

[120] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[121] Mark Newman. 2010. *Networks: an introduction*. Oxford university press.

[122] James Newsome and Dawn Song. 2005. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In Proceedings of the 12th Network and Distributed Systems Security Symposium*. Citeseer.

[123] Andrew Y Ng, Michael I Jordan, and Yair Weiss. 2002. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*. 849–856.

[124] Beng Heng Ng and Atul Prakash. 2013. Expose: Discovering potential binary code re-use. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 492–501.

[125] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. 2016. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE),*. IEEE, 780–785.

[126] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–A Locality Sensitive Hash. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth*. IEEE, 7–13.

[127] Ketan Patil and Aditya Kanade. 2018. Greybox fuzzing as a contextual bandits problem. *arXiv preprint arXiv:1806.03806* (2018).

[128] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[129] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 709–724.

[130] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 406–415.

[131] Protean Security. 2018. Next Generation Dynamic Analysis with PANDA. (2018). https://bit.ly/2ZfXlq

[132] Jing Qiu, Xiaohong Su, and Peijun Ma. 2016. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering* 42, 2 (2016), 187–202.

[133] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[134] Sanjay Rawat and Laurent Mounier. 2012. Finding buffer overflow inducing loops in binary executables. In *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 177–186.

[135] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Security Symposium*. 861–875.

[136] Nathan E Rosenblum. 2011. *The Provenance Hierarchy of Computer Programs*. Ph.D. Dissertation. University of Wisconsin–Madison.

[137] Paria Shirani, Leo Collard, Basile L Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. 2018. BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 114–138.

[138] Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2017. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 301–324.

[139] Yan Shoshitaishvili. 2017. *Building a Base for Cyber-autonomy*. Ph.D. Dissertation. University of California, Santa Barbara.

[140] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*.

[141] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 138–157.

[142] Maksim Shudrak. 2017. WinHeap Explorer: Efficient and Transparent Heap-Based Bug Detection in Machine Code. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 94–101.

[143] Steelix. 2020. LAVA-M. (2020). https://sites.google.com/site/steelix2017/home/lav

[144] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, Vol. 16. 1–16.

[145] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *ACM Sigplan Notices*, Vol. 39. ACM, 85–96.

[146] Gaith Taha. 2007. Counterattacking the packers. *McAfee Avert Labs, Aylesbury, UK* (2007).

[147] Randy Torrance and Dick James. 2009. The state-of-the-art in IC reverse engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 363–381.

[148] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[149] Fabian Van Den Broek, Brinio Hond, and Arturo Cedillo Torres. 2014. Security testing of GSM implementations. In *International Symposium on Engineering Secure Software and Systems*. Springer, 179–195.

[150] Dong Wang, Xiaosong Zhang, Ting Chen, and Jingwei Li. 2019. Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface. *Security and Communication Networks* 2019 (2019).

[151] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 61–64.

[152] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[153] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. 2013. RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing. *KSII Transactions on Internet & Information Systems* 7, 8 (2013).

[154] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

[155] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Brad Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. *arXiv preprint arXiv:1801.02742* (2018).

[156] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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, 363–376.

[157] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 359–368.

[158] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. 2019. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2525–2527.

[159] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium USENIX Security 18)*. USENIX Association, 745–761.

[160] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*.

[161] Michal Zalewski. 2010. American Fuzzy Lop: a security-oriented fuzzer. *URl: http://lcamtuf.coredump.cx/afl/* (2010).

[162] Junyuan Zeng, Yangchun Fu, Kenneth A Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 487–498.

[163] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*. 337–352.

[164] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1099–1114.

[165] Chaoshun Zuo, Wubing Wang, Rui Wang, and Zhiqiang Lin. 2016. Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*.

[166] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).