Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies

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

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, the underlying software (a.k.a. firmware) very often suffers from a wide range of security vulnerabilities, mainly due to their outdated systems or reusing existing vulnerable libraries; which is evident by the surprising rise in the number of attacks against embedded systems.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, and hybrid approaches. Furthermore, we perform both quantitative and qualitative comparisons among the surveyed approaches. Moreover, we devise taxonomies based on the applications of those approaches, the features used in the literature, and the type of the analysis. Finally, we identify the unresolved challenges and discuss possible future directions in this field of research.

CCS Concepts: • **Security and privacy** → *Vulnerability management*; **Embedded systems security**; *Software reverse engineering*; • **Computer systems organization** → **Firmware**; • **Computing methodologies** → *Machine learning*;

Additional Key Words and Phrases: Binary code analysis, embedded device security, firmware analysis, internet of things (IoT), vulnerability detection

Abdullah Qasem and Paria Shirani contributed equally to this research.

We would like to thank the anonymous reviewers for the invaluable comments. This research is the result of a fruitful collaboration between members of the Security Research Centre (SRC) of Concordia University, Hydro-Québec, and Thales Canada under the NSERC/Hydro-Québec/Thales Senior Industrial Research Chair in Smart Grid Security: Detection, Prevention, Mitigation and Recovery from Cyber-Physical Attacks.

Authors' addresses: A. Qasem, P. Shirani, M. Debbabi, and L. Wang, Concordia University, 1455 Boulevard de Maisonneuve O, Montreal, Canada, H3G 1M8; emails: a_qas@encs.concordia.ca, p_shira@encs.concordia.ca, {debbabi, wang}@ciise.concordia.ca; B. Lebel, Thales Canada Inc., 2800, av Marie-Curie, Saint-Laurent, Montreal, Canada, H4S 2C2; email: Bernard.LEBEL@ca.thalesgroup.com; B. L. Agba, Institut de recherche d'Hydro-Québec, 1800 Boulevard Lionel-Boulet, Varennes, Montreal, Canada, J3X 1S1; email: Agba.BasileL@ireq.ca.

0360-0300/2021/03-ART25 $15.00
https://doi.org/10.1145/3432893

ACM Reference format:
Abdullah Qasem, Paria Shirani, Mourad Debbabi, Lingyu Wang, Bernard Lebel, and Basile L. Agba. 2021. Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies. *ACM Comput. Surv.* 54, 2, Article 25 (March 2021), 42 pages.
https://doi.org/10.1145/3432893

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 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 [10]. 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 [71]. In 2014, Black Energy [114] APT caused a blackout that occurred by taking control of operating stations. Similar threats have been demonstrated in other countries, for instance, taking control of over 50 power generators by intruders may potentially affect the power supply to 93M U.S. residents [143]. 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, 132, 135]. 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 [79], 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 make 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., References [68, 80, 83, 87, 151]) 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 embedded systems and firmware images is even more challenging than normal binary analysis, since in addition to the challenges related to normal binary analysis, there exist specific challenges to analysis of embedded systems. 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) that cause memory corruption can be more easily detected, since the device crashes. Finally, firmware reverse engineering that involves firmware acquisition, unpacking, and binary identification is a specific challenge for embedded systems.

Even though there are a few efforts to examine 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 [23] 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. [82] study existing cyber reasoning systems for vulnerability detection, exploitation, and patching. Another work [84] surveys software vulnerability analysis, which employs machine learning techniques at the source-code level. In summary, no prior survey focuses on the vulnerability detection mechanisms for firmware and binaries in the embedded devices to identify their strengths and weaknesses and finally poses important open research questions in this domain.

In this survey, we carry out a comprehensive study of binary analysis approaches proposed for automatic inspection of vulnerabilities in embedded devices and firmware images. We systematically compare 69 individual works related to embedded systems and firmware analysis and examine about 200 papers in this area, including relevant publications in major conferences and journals in both cybersecurity and software engineering from January 2004 to January 2020. The former includes the 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). The latter includes the 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 enumerate both generic and embedded-specific binary analysis challenges and techniques for automated vulnerability detection, and then categorize the approaches based on their application domains (Section 2). In addition, we examine several existing dynamic analysis and symbolic execution works as well as static solutions to gain more insights into the proposed techniques, their strengths, and limitations (Sections 3 and 4). We devise a taxonomy based on three types of analysis employed in the existing approaches in embedded systems, including dynamic analysis, symbolic execution, and static solutions (Section 3). We also study and categorize the existing features that are being used in binary analysis in embedded systems (Section 4). Furthermore, we compare those approaches based on different criteria, i.e., methodologies, implementations, and evaluations in the corresponding sections. Finally, we discuss the lessons learned from this survey and also present the future research directions (Section 5).

We observe that most of the existing works perform their evaluations 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 different analysis techniques (e.g., static and dynamic), could be more suitable for vulnerability detection on embedded systems.

The main contributions of this article 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 2 provides background on binary analysis for embedded systems. Sections 3 and 4 survey existing dynamic analysis, symbolic executing, and static approaches for embedded systems, respectively. Section 5 highlights the lessons learned and future directions. Section 6 concludes the article.

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 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 runtime 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 [157]. Additionally, in some cases such as *binary-blob* firmware [135], the entry point and the base address are not known. Moreover, some functions may have multiple entry points [16], 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 [16]. Moreover, some functions may not be continuous and have some gaps including data, jump tables, or instructions from other functions [16], 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 that 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, 149]. Similarly, malware authors apply some techniques on their malicious code to evade analysis and make them more cumbersome. Obfuscation [35, 96, 156], encryption, and packing [90, 141] 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 [123]. 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 [134].
- **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 interconnectivity between them. As such, vulnerability detection at large scale 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 [109]. 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 [109] 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 worse if it is embedded in an operational technology (OT), e.g., triggering the brake actuator of a vehicle. Silent memory corruption in an embedded system is a rule, not an exception compared to traditional desktops [109]. 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 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 [142]. 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 [74], BAT [76], 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 [78], FILE [50], and BINWALK [74]. There exist some types of firmware that have no underlying operating system (OS). 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 runtime environment and loading the binary is more challenging [135].

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 [134], or they cannot find all the vulnerabilities (e.g., runtime vulnerabilities) that generate 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.

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 behavior that helps to identify the existence of vulnerabilities. In general, fuzzing approaches suffer from the C10 limitation. Applying fuzzing to embedded systems is more challenging than traditional computing environment. For instance, memory corruption detection is more limited due to missing fault detection techniques (C11). And, 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 [109] shows that liveness checking is insufficient to capture various

vulnerability types in the absence of source code when the memory is corrupted. In addition, embedded platform faces an additional challenge. Those devices may be triggered to commit a hard reset and device wipe through some commands and recovery might not be an option if that happens, as opposed to the normal binary world where a snapshot from a virtual machine can be restored and relaunch a new acquisition if the worst happens.

- *Instrumentation*: Instrumentation is a technique to collect and insert execution feedbacks (e.g., system calls, runtime information, and crash harness functionalities) to analyze the behavior of an application during the execution. Instrumentation information may be obtained at compile-time if the source code is available, or at runtime. Since the source code is rarely available, runtime instrumentation can be employed to re-write the binary and inject instrumentation information. However, runtime instrumentation is currently available only for embedded systems that have OS and are compatible with current emulation. These 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 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 [154], and being very slow [109]. 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., Reference [17]) 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 (C10 and C11), as mentioned earlier [109]. During symbolic execution, state exploration proceeds until a crash is detected or a termination condition is met, otherwise the

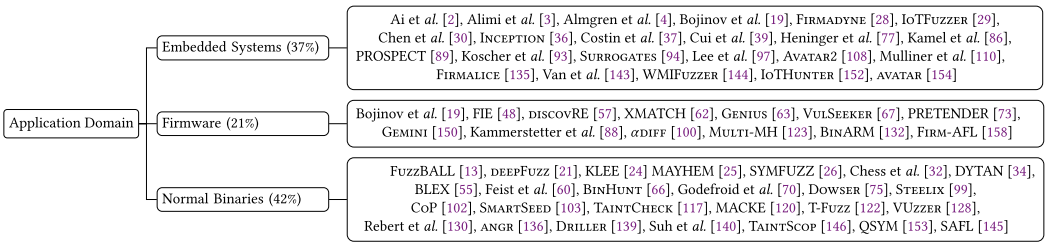

Fig. 1. Taxonomy of vulnerability detection domains.

execution continues. Thus, if corruptions are not detected, symbolic execution spends the time on worthless states. Therefore, symbolic execution also has C10 and C11 limitations.

2.3 Taxonomy of Application Domains

In this section, we categorize the existing vulnerability detection works according to their application domains. The proposed categorization, includes the three *Embedded Systems*, *Firmware*, and *Normal Binaries* domains, as presented in Figure 1. An embedded system is the combination of hardware and sufficient software to perform the desired functionality, whereas firmware is the code embedded in a hardware. As such, we group the approaches that are only focused on the analysis of the firmware into the "Firmware" category, while those that also consider any interaction with the hardware are categorized as "Embedded Systems." We also provide the distribution of these approaches over the selected works based on their relevance to the topic (i.e., vulnerability detection in embedded devices and firmware), publication years (January 2004 to January 2020) and venues (e.g., top-tier cybersecurity and software engineering conferences and journals), as discussed in Section 1. As seen, 37% of the works have been applied on the embedded systems and 21% of them on firmware images, which highlights the importance of proposing novel solutions applicable to this domain.

2.4 Embedded Device Classification

As explained earlier, an embedded system is composed of hardware and software designed to solve a specific task that may involve interacting with the environment through sensors and actuators. A wide range of devices, such as digital cameras, printers, hard disk controllers, smartphones, automobiles, and smart meters 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. To perform a comparative study among existing works, we aim at classifying the embedded devices. Since we focus on vulnerability detection in embedded systems, we follow the classification proposed by Reference [109], which is based on the OS type. 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 devices 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, such as access points and routers.
- *Type-II: Embedded OS-based.* Type-II OSs are designed for embedded devices with low computation resources, such as CPU, memory or power. *Type-II* OS may not have a Memory Management Unit (MMU) as present in 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.

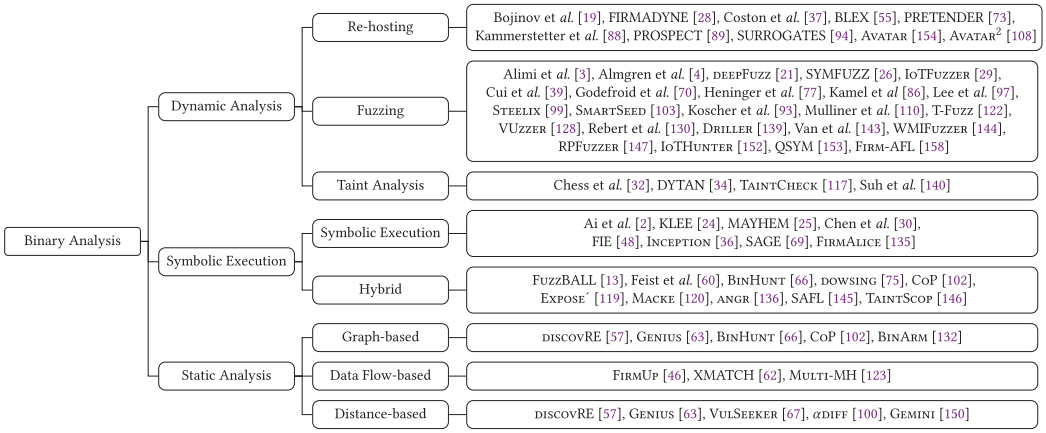

Fig. 2. Taxonomy of binary analysis approaches.

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

3 DYNAMIC ANALYSIS AND SYMBOLIC EXECUTION

In this section, we provide a detailed review of dynamic analysis and symbolic execution approaches applied to embedded systems. Our review is organized around our proposed taxonomy, shown in Figure 2. We further summarize the traditional approaches for normal binaries that have been already applied to the embedded system environment. Moreover, we provide a comparative study on the existing approaches applied on embedded systems as well as on the traditional approaches that can potentially be employed in embedded systems.

3.1 Dynamic Analysis on Embedded Systems

In this section, we review dynamic analysis approaches that employ re-hosting and fuzzing on embedded systems. We also briefly introduce existing approaches on normal binaries that can potentially be utilized for the embedded systems.

3.1.1 Re-hosting. Embedded systems encounter specific scalability issues compared to regular platforms. In a traditional computing environment, parallel techniques using multi-processing or virtualization can accelerate testing campaigns. Conversely, 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 section, we outline the existing contributions that perform partial/full emulations and web interface extraction by using re-hosting.

Partial Emulation. Analyzing embedded systems in a virtual and scalable environment is a challenging task, since running firmware requires, every so often, 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 as necessary. Partial emulation offers the impression of having a full emulation of the targeted embedded system with all its peripherals. However, its scalability is affected by the frequent interactions with the connected peripherals (C9). Also, scaling across multiple devices for multiple simultaneous testing can be problematic.

Re-hosted embedded system firmware regularly need access to their peripherals to finish their testing. Therefore, to facilitate the adaption of dynamic binary analysis approaches over re-hosted embedded devices, Kammerstetter et al. [89] propose a partial emulator called PROSPECT. PROSPECT implements a proxy to tunnel every peripheral hardware access initiated from the firmware running inside the QEMU[1] [54] virtual machine to the embedded device under the test. To assess this solution, PROSPECT is evaluated over a commercial fire alarm system by running it on the environment setup, and then an extensive mutation-based fuzzing is conducted using an original stream of captured network traffic packets. Therefore, no network protocol specification is required in advance. Fuzzing is carried out by frequently modifying one byte randomly each time. As a result, a zero-day vulnerability is discovered on targeted devices. However, PROSPECT cannot be applied to the devices that have no network connection interfaces.

Similar to PROSPECT [89] but more scalable, a dynamic analysis framework named Avatar[2] [154] 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 their peripherals, Avatar[2] [108] (a follow-up work of avatar) can orchestrate interactions with other physical devices, debuggers, and popular binary analysis frameworks, such as angr[3] [136] (explained in Section 3.3), QEMU, PANDA[4] [125], and OpenOCD [53]. Currently, Avatar[2] supports x86, x86-64, and ARM hardware architectures, nevertheless, its modularity feature allows its adaption 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 that require near-real-time response. To address this issue, a partial emulator called SURROGATES [94] 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 [154] and more suitable to test time-sensitive and complex systems, such as medical devices.

In contrast to the aforementioned emulation approaches, Kammerstetter et al. [88] 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 sufficient training with a connected device. Then, the physical devices are no longer needed for repeated testing. Since this approach offers snapshotting

[1] https://www.qemu.org/.
[2] https://github.com/avatarone/avatar-python, http://www.s3.eurecom.fr/tools/avatar/.
[3] https://github.com/angr/angr.
[4] https://github.com/panda-re/panda.

and parallelization, advanced dynamic analysis can be employed. However, the authors have shown that their peripherals caching approach can only be applied to small complex firmware. They have also shown that the proposed approach suffers from state explosion similar to symbolic execution.

Full Emulation. To use dynamic analysis over embedded devices in a scalable manner compared to traditional computing devices, re-hosting targeted embedded systems firmware outside their devices is required. This can involve either an emulation or a virtualized environment without the need to have access to any peripherals. This process is called full emulation, which is more scalable than partial emulation. Also, it does not require the physical existence of the targeted embedded system hardware. However, full emulation is only applicable when peripherals of the targeted system can be successfully emulated. Full emulation is practical when the embedded system is Linux-based and its firmware is successfully extracted [37] (C12) or when a full documentation of the targeted embedded system hardware is available. Full emulation opens the door to adopt scalable dynamic analysis approaches into embedded system world.

To achieve this goal, Chen et al. [28] introduce a full-emulator dynamic analysis framework named FIRMADYNE[5], which is specialized in Linux-based firmware embedded systems. 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, 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 and symbolic execution) more scalable and applicable on embedded system firmware running in a full-emulation or virtual environment, Gustafson et al. [73] propose the PRETENDER framework, which automates the re-hosting of the various embedded systems' firmware in a virtual environment. As a result of automated re-hosting, dynamic analysis solutions could 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. Afterwards, the resulting models are integrated either with a well-known full emulator (e.g., QEMU) or in a framework analysis (e.g., ANGR [136]) for further interactive and precise analysis in a scalable manner for its related firmware. PRETENDER was evaluated on six distinct "blob" images on three different platforms where each target contains synthetic security vulnerabilities. Moreover, each target is tested by using naive fuzzing and code coverage to cover as much as of the targeted firmware functionalities. PRETENDER's goal is not to find new vulnerabilities in the targeted embedded systems; rather, it enables an advanced dynamic analysis approach to find known vulnerabilities.

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. Costin et al. [37] propose a scalable and automated dynamic analysis framework to discover vulnerabilities in firmware that use a web management 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

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

from the firmware, then it applies static analysis using RIPS[6] [41] to find potential vulnerabilities in the PHP web interface of the targeted embedded system. Moreover, it runs the web interface in an emulated environment. Finally, the running web interfaces are tested using the open-source web penetration testing Arachni[7], zed attack proxy (ZAP)[8], and w3af[9] tools. As a result, 225 serious vulnerabilities were discovered and verified. The proposed system [37] is the first one that is aimed at the security of PHP in embedded web interfaces.

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

Unlike the aforementioned approaches, Bojinov et al. [19] investigate the embedded systems web interface manually. The authors demonstrate that all of the 21 investigated embedded system devices contain critical web vulnerabilities, e.g., cross channel scripting (XCS). XCS utilizes multiple features included in modern embedded devices, such as cross-channel interactions between web interface and FTP server on network-attached storage (NAS) or the interaction between the web server and the SIP phone service to compromise the embedded devices web interface. The authors successfully report around 50 vulnerabilities across 16 different vendors.

3.1.2 Fuzzing. Re-hosting the firmware outside its environment has several challenges (C5, C9, C12). To overcome these challenges, fuzzing approaches still could be applied over a communication interface when re-hosting is not possible. To this end, targeted embedded systems are tested over a communication interface using fuzzing. Fuzzing approaches can be divided into three categories: *mutation-based*, *generation-based,* and *hybrid*. Based on the dependency of the fuzzing approach relative to the targeted program code, these techniques could be further categorized into three groups [104]: (i) *Black-box* fuzzers, which generate input seeds without the need to have access to the internal state of the targeted program. (ii) *White-box* fuzzers, which generate input seeds by considering both input coverage and targeted program internal analysis. (iii) *Gray-box* fuzzers (also known as coverage-based fuzzer), which mutate generated input seeds based on the instrumentation feedback that monitors code coverage resulted from each input seed. The *gray-box* fuzzers fall in between black-box and white-box to increase the speed of the fuzzing process. Also, fuzzers can be categorized into *directed fuzzing* and *coverage-based* fuzzing categories. The objective of the former is to produce test cases covering targeted program code and paths while performing a fast testing process. However, the goal of the latter is to produce test cases that cover as much as possible of the targeted software/firmware. These fuzzers intend to accomplish a more thorough testing to discover as many bugs as possible [98]. In the following, we outline those solutions that have been applied on the embedded devices.

Mutation-based Fuzzers. Mutation-based fuzzers are easy to implement and use, and therefore they have been utilized by different state-of-the-art fuzzers. In this setup, test cases are generated by using random mutation techniques, which in essence 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 its behavior waiting for an error or a crash that might be triggered by one of the mutated inputs. The mutation-based fuzzers can be applied in black-box manner or in coverage-based manner.

[6] https://github.com/robocoder/rips-scanner.
[7] http://www.arachni-scanner.com/, https://code.google.com/p/zaproxy/, http://w3af.org/.
[8] https://owasp.org/www-project-zap/.
[9] http://w3af.org/.

Chen et al. [29] propose an IoT fuzzing framework named IoTFuzzer to locate memory corruptions. First, IoTFuzzer leverages embedded systems–related mobile apps that are used to control them remotely. IoTFuzzer performs dynamic analysis on the target firmware mobile app to derive the logic of the command messages. This allows identifying command format, sent URLs, 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 protocols. IoTFuzzer was evaluated on 17 real-world IoT devices and found 15 vulnerabilities, 8 of which have never been reported before. However, IoTFuzzer cannot provide the location of a vulnerability; rather, it only reports the input that triggers it.

While IoTFuzzer shows promising results over fuzzing IoT firmware, it may be unsuitable in the case of complex stateful messaging protocols (e.g., SNMP, FTP, SSL, BGP, and SMB), which maintain strict validation mechanisms (e.g., checksum and message length) for almost every received message. Hence, malformed inputs will diverge from current protocol state and thus miss the chance of discovering deep bugs. To address stateful network protocols, Yu et al. [152] propose the IoTHunter, which implements a new technique using multi-stage message generation, and is integrated with American Fuzzy Lop (AFL) [155] (explained later in this section), Boofuzz [20], and Avatar^2. 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 gray-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 provides better results compared to black-box fuzzer Boofuzz [20] with respect to edge coverage, block coverage, and function coverage.

Zheng et al. [158] 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], since the latter already solved many hardware compatibility issues compared to other emulators, such as Avatar [154]. Firm-AFL also solves 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.

Alimi et al. [3] propose a mutation-based fuzzing approach using a genetic algorithm to generate test-inputs for examining the MasterCard 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 MasterCard 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 leads to those illegal transactions.

Fuzzing has been applied to investigate potential vulnerabilities in the modern automobile systems. Modern automobiles are controlled by various computerized and networked systems. Koscher et al. [93] perform a comprehensive investigation over two 2009 automobiles in a controlled lab and real road. They implement a system to sniff and fuzz the packets sent into the vehicle CAN bus, and they successfully generate packets to perform various functionalities, including unlocking doors, triggering the horn, turning off the lights, and so on. Moreover, they demonstrate crafting attacks with the capability to endanger the driver and passengers (e.g., suddenly disengaging the brakes while driving). Similarly, Lee et al. [97] 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.

Generation-based Fuzzers. Generation-based fuzzers require the input format of the targeted program to generate the testcases. Testcases made via 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 this limitation of random mutation techniques by restricting the generated inputs to a specific data structure or grammar to ensure reaching the deep level of a program.

Similar to IoTFuzzer, WMIFuzzer [144] tests a 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 nodes of the tree contain the contents of the messages fields. Thus, alteration will be applied only on the tree nodes to ensure valid message construction. The final step performs the fuzzing over the altered messages. Network monitoring techniques are implemented to detect if a vulnerability was triggered. Since not all the triggered vulnerabilities will crash the target device, the altered message will contain injected *reboot command* and *Interface Leak*. Then, their side effects–altered messages are monitored from the observed network. WMIFuzzer was evaluated on seven COTS popular IoT devices, and it discovered 10 vulnerabilities, 6 of which are zero-day ones. The limitation of WMIFuzzer is that it works only with IoT devices that have a web management interface accessed over either HTTP or HTTPS.

The RPFuzzer [147] introduces a predefined expert mathematical model tailored for the communication protocol to generate the initial messages (seeds) to be subsequently used as starting points during the fuzzing campaign. Then, mutation-based fuzzer is applied on the generated seeds for test case generation. To detect if a vulnerability is triggered, three monitoring methods are used: sending normal packets, monitoring CPU utilization, and checking system logs. RPFuzzer was evaluated on Cisco routers where test runs are launched over the SNMP protocol. RPFuzzer discovered eight vulnerabilities, five of which are zero-day.

There exist smart cards that implement web technologies to facilitate the communication with other networks. Kamel et al. [86] implement a black-box-based fuzzer to investigate the potential vulnerabilities of web servers embedded in the smart cards. The authors further test whether the implemented HTTP server is compliant with the design specification. The evaluation results demonstrate that some smart card HTTP servers accept administration commands that should be available only to the card issuer. Furthermore, several non-compliant behaviors have been discovered.

Fuzzing was also 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. Their experiments demonstrate that the generated input packets can uncover potential denial of service vulnerabilities and in some cases can also corrupt the PLC execution. To test smart phones adherence to the GSM specification, Mulliner et al. [110] and Van et al. [143] design generation-based fuzzers that can trigger reboot, memory exhaustion, and denial of service.

However, to evaluate the security of embedded devices connected to the internet, Cui et al. [39] perform global network scanning. The authors utilize *Nmap*[10] and scan both telnet and HTTP

[10] https://nmap.org/book/man-port-scanning-techniques.html.

ports to detect firmware devices that can be accessed 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. [77] perform network scanning on TLS and SSH protocols. The authors demonstrate that the majority of embedded system devices are using weak RSA key generation algorithms that have low entropy. Furthermore, it is shown that some embedded systems are using predictable DSA private keys.

Fuzzing Approaches in Normal Binaries. In traditional computing environments, fuzzing techniques have been used more broadly, sometimes combining static and dynamic analysis. However, such approaches have not been applied to embedded systems yet. We briefly introduce some fuzzing approaches that are used on normal binaries as follows:

There exist some works that employ random mutation to fuzz normal binaries. These include AFL [155], which is a simple and efficient gray-box fuzzer to detect vulnerabilities; Rebert et al. [26, 130], which propose mathematical solutions to efficiently select seeds for any types of fuzzer within a constrained time and computing budget to increase the number of discovered bugs; and SYMFUZZ [26], which allows to maximize bug discovery in black-box fuzzing. Moreover, instead of mutating the input seed to cover more execution paths within the targeted program, T-Fuzz [122] transforms the targeted programs to accept the provided input seeds by removing or disabling the sanity checks, which would otherwise prevent the provided seeds from advancing on various execution paths.

In the field of Grammar-based fuzzers, Godefroid et al. [70] utilize neural network to learn the structured inputs of a program from samples to generate well-structured, diverse, and high-coverage input file seeds to fuzz the targeted program. Afterwards, fuzzing is applied to modify the structure of the generated seeds to increase the chance of reaching unpredictable code paths where bugs or vulnerabilities may be encountered. Similarly, SmartSeed [103] utilizes generative adversarial networks (WGAN) [12] to train a model from selected samples for generating seeds in multiple formats (e.g., mp3, bmp, and flv). Moreover, it can learn to generate corrupted seed files to be employed to trigger more crashes. SmartSeed has been designed and tested to be compatible with other fuzzers, such as AFL.

To address the limitations of fuzzing approaches and to increase the probability of locating more potential bugs and vulnerabilities, fuzzing approaches have been integrated with other binary analysis approaches to help bypass conditional branch checking either on a concert or a magic value. For example, Driller [139] combines both concolic execution and fuzzing and runs them in a repeated and alternate manner. It starts by fuzzing the targeted binary until it gets stuck due to a complex conditional path check (e.g., 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. Thus, Driller updates the fuzzer with new inputs to continue the exploration, and it repeats this procedure until a crash occurs in the target application. Similar to Driller, deepFuzz [21] combines concolic execution with constrained fuzzing, but it gives probabilities to each path resulting from the symbolic execution solver. Then, the most likely paths will be further explored by the fuzzer. Furthermore, SAFL [145] uses components similar to those used by Driller, except the oscillation between fuzzing and symbolic execution, since it is costly. SAFL uses symbolic execution only once to create good initial seeds and then continues with fuzzing guided by an efficient mutation algorithm. However, VUzzer [128] combines fuzzing with dynamic taint analysis (DTA). To guide the fuzzer towards reaching the deeper paths while maximizing the coverage, DTA with static analysis is utilized to extract data flow and control flow features. Similar to VUzzer [128]

and Driller [139], Steelix [99] utilizes lightweight program analysis by performing both static analysis and binary instrumentation.

3.2 Symbolic Execution on Embedded System

Symbolic execution is another avenue that has been utilized to tackle the embedded system hardware compatibility challenge (C5) and 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 the outcomes of peripherals accesses as symbolic data. Using symbolic execution improves scalability and helps with exploring more code paths. However, it has its own limitations (explained in Section 2), such as path explosion, which could be amplified when the firmware requires access to its peripherals (e.g., interrupts generated from the peripherals frequently create more states [15]).

Symbolic Execution. One of the early proposed approaches, called FIE [48], introduces symbolic execution to the 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 [24] (explained in Section 3.3) 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 symbolic execution. It successfully identified 21 bugs, and its scalability is tested by a system built on Amazon EC2 [9].

A cross-architecture framework called Firmalice [135] investigates the existence of various 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 a *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 extracts 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 [24], MAYHEM [25], and FuzzBALL [13] (explained in Section 3.3), 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 trigger the vulnerability. Firmalice reported the existence of a backdoor in two commercial firmware devices.

Another symbolic execution framework proposed for testing embedded system firmware is Inception [36]. It is built on top of KLEE [24], which requires access to the source code. It leverages the KLEE [24] to solve the interrupts issues, resulting 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 introduces JTAG debugger. Moreover, it includes a translator to lifted source code using LLVM and merges it with potential handwritten assembly instructions by developers. Inception has been evaluated over both synthetic and real-world systems, 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, since it requires powerful computation resources, in terms of both CPU and memory, which are typically limited in embedded systems. Moreover, symbolic execution depends upon 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 powerful remote hosts, such as personal computers, workstations, and so on. To establish this coordination, cross-debugging functions provided by the embedded system vendor (i.e., 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, which impacts its scalability.

Similar to Reference [30], Ai et al. [2] introduce concolic execution into embedded system testing using the same methodology in Reference [30] while supporting multiple architectures, such as x86, ARM, and PPC. To achieve their objective, the authors utilize a portable instrumentation scheme. They lift firmware and state information into the VEX intermediate representation to make it easy for the symbolic engine to deal with constraints in multiple CPU architectures.

Symbolic Execution and Taint Analysis in Normal Binaries. In normal binary computation, symbolic execution has been integrated with other techniques, such as static analysis and taint analysis. For instance, MACKE [120] integrates static analysis with symbolic execution to uncover buffer overflow vulnerabilities and to report their severity scores. Similarly, Feist et al. [60] integrate static analysis with symbolic execution to excavate and prove the existence of use-after-free vulnerabilities (UAF) in binary code. Another example includes TaintScop [146], which is used to bypass checksum checks by integrating symbolic execution with taint analysis. In addition, dynamic taint analysis (DTA) has been employed solely to identify the vulnerabilities in normal binaries. For instance, Chess et al. [32] apply DTA to locate input validation vulnerabilities by leveraging the calculation devoted to functional testing. DYTAN [34] performs data flow and control flow taint analysis on source code or binaries that might be stripped. Moreover, to automatically detect overwrite attacks and generate corresponding signatures, TaintCheck [117] identifies tainted data. However, Suh et al. [140] utilize DTA to protect programs from malicious inputs at OS level.

3.3 Reused Traditional Approaches in Embedded Systems

Different binary analysis approaches have been extensively evaluated on traditional computing systems, such as desktops. Although the main focus of such approaches is not on embedded system devices, some of these approaches have been already introduced or applied on 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 [136] (black-box mutation fuzzer and white-box fuzzer) and AFL [158] (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 have significantly improved the usability of dynamic analysis over embedded system devices. In this section, we summarize the existing approaches that have been already reused in the embedded system domain.

The popular symbolic execution tool called KLEE [24] automatically generates test cases with high execution path coverage for both simple and complex software programs. KLEE aims to reach every executable instruction, and it investigates critical operations that could be potentially exploited by the attackers. Therefore, KLEE implements different constraint-solving optimizations (e.g., expression rewriting) to improve efficiency, to decrease the solving time and to reduce memory use. To this end, KLEE maintains a compact program state through implementing *copy-on-write* at the object level and 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 is evaluated over *GNU coreutils* and successfully reported 56 serious bugs in 452 tested applications, and achieves an average code coverage of 90%. KLEE is utilized by different approaches that work over the source code of available firmware, as explained in Section 3.2. Specifically, it is utilized by FIE [48], Firmalice [135], and Inception [36] to extensively cover different possible execution paths.

To facilitate the integration of various binary analysis approaches, a systematized open-source binary analysis framework named angr [136][11] is introduced. It reproduces a set of state-of-the-art binary analysis approaches implementations in a single and coherent framework. Furthermore, angr re-implements some offensive binary analysis, such as automatic exploit generation, ROP shellcode, exploit replay, and so on. 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 advantage and compensates for each approach limitation. angr lifts the binary program into VEX-IR to support various CPU architectures. Angr has been integrated with several dynamic approaches for embedded systems, such as Avatar [154] and PRETENDER [73], presented in Section 3.1.1.

Babic et al. [13] propose a three-stage process approach to automate test case generation that covers more execution paths in a given binary program. The first stage utilizes both static analysis and dynamic analysis to generate inter-procedural CFG, which consist of a CFG of a function and its call graph, and represented by using visibility push-down automaton (VPA) [8]. In the second stage, static analysis is utilized to overcome false positives resulting from the first stage. Finally, symbolic execution is used to automatically generate test cases needed to trigger potential vulnerabilities based on recommendations, such as weighted shortest-path lengths in the VPA, provided by the previous two stages. The approach implementation is available as the prototype tool FuzzBALL[12]. The idea of FuzzBALL has been used in Firmalice [135], a symbolic execution approach for firmware and embedded devices.

Symbolic execution can be applied to dynamically craft the required inputs to exploit and run discovered vulnerabilities. To achieve this goal, 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. It is applied on 29 applications and successfully reports and generates 29 exploits automatically. However, MAYHEM 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. Moreover, 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 UAF. The idea of MAYHEM has been utilized in Firmalice [135].

We refer the reader for more details on different binary analysis techniques to other surveys on fuzzing [98, 104] and symbolic execution [15].

3.4 Comparative Study

In this section, we compare the existing approaches for embedded systems as well as the traditional approaches that can potentially be applied to embedded systems. We further discuss our key observations from this comparative study.

[11] https://github.com/angr/angr.
[12] https://github.com/bitblaze-fuzzball/d-s-se-directed-tests.

Table 1. A Comparison of State-of-the-art Embedded System Vulnerability Detection Approaches

Proposal	Venue	Methodology								Target		Device Type			CPU Arch.			Reused Framework(s)	Release	
		Partial Emulation	Full Emulation	Symbolic Execution	Fuzzing	Web Interface Check	Network Scanning	Taint Analysis	Static Analysis	Firmware	Embedded Devices	I	II	III	x86-64	ARM	MIPS		Open Source	Open Service
Koscher et al. [93]	S&P 2010				•						•			•	•	•	•			
Cui et al. [39]	ACSAC 2010				•		•				•	-	-	-	-	-	-			
Mulliner et al. [110]	USENIX 2011				•						•		•		-	-	-			
Heninger et al. [77]	USENIX 2012				•		•				•	-	-	-	-	-	-			
Kamel et al. [86]	IJINS 2013				•						•	•			-	-	-			
FIE [48]	USENIX 2013			•						•			•	•		*		KLEE		
RPFuzzer [147]	TIIS 2013				•						•		•		•	•	•			
Almgren et al. [4]	CRISALIS 2014				•						•		•			•				
Van et al. [143]	ESSoS 2014				•						•		•		-	-	-			
PROSPECT [89]	ASIACCS 2014	•								•	•	•				•	•	QEMU		
Avatar [154]	NDSS 2014	•								•	•		•	•		•		QEMU	•	•
Alimi et al. [3]	HPCS 2014		•		•						•			•	-	-	-			
Chen et al. [30]	TR 2014			•									•	•	•					
Lee et al. [97]	AINA 2015				•						•			•	-	-	-			
Firmalice [135]	NDSS 2015			•					•	•		•	•		•	•	•	KLEE, MAYHEM, FuzzBALL		
SURROGATES [94]	WOOT 2015	•								•	•	•		•		•				
FIRMADYNE [28]	NDSS 2016		•			•				•	•	•				•	•	QEMU		
Kammerstetter et al. [88]	SECUWARE 2016	•								•	•	•					•	QEMU		
Costin et al. [37]	ASIACCS 2016		•							•	•	•			•	•	•		•	•
Avatar^2 [108]	BAR 2018	•								•	•		•			•	•	angr, PANDA	•	•
Inception [36]	USENIX 2018			•									•	•		•		KLEE, LLVM, JTAG		
IoTFuzzer [29]	NDSS 2018				•			•			•	•	•	•	•	•	•			
IoTHunter [152]	CCS 2019		•		•						•	•	•		•	•	•	AFL, Boofuzz, Avatar2		
WMIFuzzer [144]	SCN 2019				•	•					•		•		•	•				
Firm-AFL [158]	USENIX 2019	•	•		•					•		•				•	•	Firmadyne, AFL	•	
PRETENDER [73]	RAID 2019		•							•		•				•	•	angr,QEMU		
Ai et al. [2]	ICCSP 2020			•						•	•	•			•	•		SE, VEX-IR	•	
Distribution	NA.	22%	22%	15%	52%	7%	7%	4%	4%	44%	78%	44%	48%	33%	33%	63%	44%	NA.	18%	11%

(•) means that the approach offers the corresponding feature, otherwise it is empty. (–) means that the information is not provided. (*) means the proposal supports the Microcontroller family MSP430. The "Distribution" presents the percentage of each feature category that is used in all surveyed proposals. The gray cells are for the sake of readability to separate different categories.

3.4.1 Comparing Existing Approaches for Embedded Systems. Due to the absence of similar setup environments, 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 for each solution in terms of the approaches, implementations, and evaluations as follows:

The first and second columns of Table 1 specify proposals and corresponding venues ordered by the date. The next eight columns present their main proposed methodologies. The "Target"

Table 2. A Detailed Comparison of State-of-the-art Embedded System Vulnerability Detection

Approach	Challenge Overcome	Target (Sector)	Access		Protocol
			Source Code	Device	
Koscher et al. [93]	C5,C10,C11	Automotive			CAN
Cui et al. [39]	C5,C9	IoT devices		•	Telnet, HTTP
Mulliner et al. [110]	C10,C11	GSM feature phones			
Heninger et al. [77]	C5,C9	IoT devices		•	TLS
Kamel et al. [86]	C5,C11	Smart cards			
FIE [48]	C5, C9	Micro-controller FamilyMSP430	•		
RPFuzzer [147]	C5,C10,C11	Cisco routers		•	SNMP
Almgren et al. [4]	C5,C11	PLCs and smart meters			
Van et al. [143]	C10,C11	GSM feature phones			
PROSPECT [89]	C5,C9	Building automation (alarm system)		•	TCP/IP
Avatar [154]	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.[97]	C5,C11	Automotive			CAN
Firmalice [135]	C5,C9,C10	Smart meter, CCTV camera, Dell Laser Mono Printer			
SURROGATES [94]	C5,C9	Medical devices		•	
FIRMADYNE [28]	C5, C9, C11	Embedded system with network/web services (e.g., routers)			HTTP/HTTPS, Telnet
Kammerstetter et al. [88]	C5,C9	Simple firmware (GNU core utilities)		•	TCP/IP
Costin et al. [37]	C5,C9,C11	Routers and switches			HTTP
Avatar^2 [108]	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 [152]	C5,C9,C10,C11	Home router and NSA Synology			NMP, FTP, SSL, BGP, SMB
WMIFuzzer [144]	C5,C10,C11	SOHO router, IP camera, gateway		•	HTTP/HTTPS
Firm-AFL [158]	C5,C9,C10,C11	Routers			HTTP, SSH
PRETENDER [73]	C5, C9	ARM mbed			
Ai et al. [2]	C5,C9,C10	FriendlyARM mini2440, Kyocera 8000 series printer, binutils programs		•	

(•) means that the approach offers the corresponding feature, otherwise it is empty.

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 the next three columns, we mark which CPU architectures are supported by these approaches. The next column indicates which framework(s) have been used by the approaches. 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 the distribution of these features (in the last row of the table) over the selected works on dynamic analysis and symbolic execution, based on their relevance to the topic (i.e., vulnerability detection in embedded devices and firmware), publication years (January 2004 to January 2020) and venues (e.g., top-tier cybersecurity and software engineering conferences and journals), as discussed in Section 1.

Moreover, more details on these approaches are 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 tested embedded

device. The next two columns specify if a given proposal needs access to the source code of the embedded device firmware and the physical device itself, respectively. The last column outlines the type of communication protocol(s) used to test the targeted embedded device by a given proposal.

Additional qualitative and quantitative comparisons for fuzzing and symbolic execution approaches in terms of their implementations and evaluations are shown in Table 3. The first column of the table specifies proposals. The second part lists the proposals based on fuzzing types including black-box, white-box, and gray-box fuzzers. We consider symbolic execution as a white-box fuzzer, following the definition provided by Reference [69]. The third category outlines the proposals based on input types and supported formats. The fourth category ("Dataset") reports the number, domain, and the public availability of the dataset used for evaluation. The fifth part provides the results reported by each proposal, such as how many vulnerabilities are discovered, how many of them are zero-day, and the number of different types of detected vulnerabilities. The sixth category specifies the platform used to run the experiments. The seventh part indicates the duration of the fuzzing campaign against the targeted dataset reported in "Dataset" columns. The two last columns report the number and the name of the approaches, which are compared with the proposal.

Key observations of our comparison. Our key observations from Table 1, Table 2, and Table 3 are as follows: One of the main objectives of various binary analyses over embedded systems is to overcome scalability issues, hardware compatibility, and applying well-known dynamic analysis approaches over traditional environments applicable in the embedded system world. Narrowing down the observations, the majority of selected approaches have been evaluated over the ARM and MIPS CPU architectures. Moreover, the proposed approaches are less likely to target embedded devices of Type-III compared to the other approaches that target embedded devices of Type-I and Type-II. Furthermore, the majority of approaches do not need access to firmware source code with the exception of two, since those works are based on KLEE [24], which works on the source code. Moreover, full emulation approaches or approaches integrated with full emulation are more scalable, since they do not need access to any hardware.

In addition, re-hosting approaches try to overcome multiple CPU architecture (C5) and scalability (C9) challenges. The fuzzing approaches try to also overcome test case generation (C10) and fault detection (C11) challenges. As can be seen, the majority of reported bugs, known vulnerabilities, and zero-day vulnerabilities are discovered by employing fuzzing approaches. Moreover, considering the features detained in both Table 2 and Table 3, fuzzing embedded systems through their provided network connection interface show a better performance by discovering more bugs and vulnerabilities with less effort compared to other approaches. However, it is difficult to inspect the location of bugs triggered during the fuzzing campaign. Another disadvantage is that fuzzing approaches require a significant amount of time; they need days to report interesting results compared to the other approaches. Finally, introducing AFL and concolic execution into an embedded system testing world opens the door to introducing various state-of-the-art approaches proposed so far for a general-purpose computing environment, such as Driller [139], TaintScop [146], and SmartSeed [103].

3.4.2 Comparing Reused and Potential Traditional Approaches for Embedded Systems. Similarly, in this section, we conduct qualitative comparisons on the traditional approaches that are reused in embedded system analysis with their competitive proposals that potentially can be introduced. This comparison is based on the available information provided for each solution in terms of the approaches, implementations, and evaluations as shown in Table 4. The first and second columns of Table 4 specify proposals and corresponding venues ordered by the date. The next seven columns present their main proposed methodologies. In the next three columns, we mark

Table 3. A Comparison of State-of-the-art Symbolic Execution and Fuzzing Vulnerability Detection Evaluations

Proposals	Fuzzing Type						Fuzzing Input				Dataset				Reported Results							OS			Comparison	
	Generation-Based	Mutation-Based	Black-Box	Gray-Box	White-Box	Coverage-Based	File	Network	Web-UI	Format	Normal Binaries	Firmware	Embedded Devices	Available	Vulnerabilities	Zero-day Vulnerabilities	Type(s) of Vulnerabilities	New Crashes	Crashes	Bugs	New Bugs	Linux	Windows	Test Campaign Duration	No. of Comparisons	Compared with
KLEE [24]					•	•					452									56	3			904 d		
SAGE [69]					•	•	•				7										30		•	70 h		
Kosher et al. [93]		•	•					•				2	2		25											
TaintScop [146]					•		•	•		png, jpeg, tiff, bmp, gif, pcap	8					6						•	•	80 m		
Kamel et *al* [86]		•	•					•		http		1			1					1				6 d		
FIE [48]					•	•						99			22		2				21			24 h		
Dowser [75]					•						6				7	2	1					•		30 m		
RPFuzzer [147]	•	•						•		Router Protocols					8	5								1440 m	3	Peach, SPIKE, Sulley
Alimi et al. [3]		•	•									1	1		1		1						•	12 h		
Rebert et al. [130]		•	•			•	•			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
Firmalice [135]												3			2		1							11 h		
Driller [139]		•			•		•				126			•	6				77					24 h	2	AFL, Mayhem
Feist et al. [60]		•			•										1	1	1							20 m		AFL, radamsa
VUzzer [128]		•		•		•	•				302								1008		8	•		30 h		
Godefroid et al. [70]	•	•	•				•			pdf	1				1	1	1						•	5 d		
Steelix [99]		•		•		•	•				22					1				272	9	•		5 h	4	FUZZER, SES, Vuzzer, AFL-lafintel
SAFL [145]		•					•				10								255					24 h	2	AFL, AFLFast
T-Fuzz [122]		•		•		•	•				300			•						166	3	•	•	24 h	3	AFL, Vuzzer, Steelix
SmartSeed [103]	•						•			mp3, bmp, flv	12				23	16	9		1096			•		72 h	1	AFL
QSYM [153]		•		•			•				12									2265	13	•		3 h	2	Driller, VUzzer
IoTFuzzer [29]		•	•					•		http, https			17		15	8	3	4	4			•		408 h	2	Sulley, BED
IoTHunter [152]		•		•		•		•		snmp, ftp, ssl, bgp, smb			8		5	2						•		30 d	2	boofuzz, IoTFuzzer
WMIFuzzer [144]		•	•					•	•	http, https			7		10	6	3		3			•		23 h	2	AFL, Sulley
Firm-AFL		•			•	•		•		http, upnp		288	7		15	2						•		50 h		
Inception [36]					•					icmp, http, uart		4		•	1,562	2	6		8							KLEE
Distribution	12%	72%	32%	2%	%	36%	48%	32%	4%	NA	56%	24%	24%	16%	64%	4%	4%	8%	32%	2%	28%	48%	2%	NA	44%	

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

Table 4. A Comparison of State-of-the-art Dynamic and Symbolic Execution Vulnerability Detection Approaches on Normal Binaries

Proposals	Venue	Methodology					CPU Arch.			Reused Framework	Release	
		Symbolic Execution	Taint Analysis	Static Analysis	Fuzzing	Network Scanning	ARM	x86-64	MIPS		Open Source	Open Service
Suh et al. [140]	NDSS 2004		•					•				
TaintCheck [117]	NDSS 2005		•					•				
DYTAN [34]	ISTR 2007		•					•				
BinHunt [66]	ICICS 2008	•		•				•				
KLEE [24]	USENIX 2008	•						•			•	•
SAGE [69]	NDSS 2008	•						•			•	
Chess et al. [32]	ISTR 2008		•					•				
TaintScop [146]	SP 2010	•	•	•				•				
MAYHEM [25]	S&P 2012	•						•				
Expose' [119]	COMPSAC 2013	•		•				•		BAP		
Dowser [75]	USENIX 2013	•	•	•				•				
Rebert et al. [130]	USENIX 2014				•	•		•				
SYMFUZZ [26]	SP 2015				•			•				
MACKE [120]	ASE 2016	•		•				•		KLEE		
Driller [139]	NDSS 2016	•			•			•		angr		
angr [136]	SP 2016	•		•	•		•	•	•		•	•
Feist et al. [60]	SSPR 2016	•		•	•			•		GUEB		
deepFuzz [21]	DIMVA 2018	•			•			•		AFL		
CoP [102]	TSE 2017	•		•				•				
VUzzer [128]	NDSS 2017		•		•			•				
Godefroid et al. [70]	IEEE/ACM 2017				•			•				
Steelix [99]	2017			•	•			•				
SAFL [145]	ICSE 2018	•			•			•		AFL		
T-Fuzz [122]	SP 2018				•			•				
SmartSeed [103]	2018				•			•				
QSYM [153]	USENIX 2018	•			•			•				
Distribution		58%	27%	35%	50%	4%	4%	100%	4%	NA	11%	8%

(•) means that the approach presents the corresponding feature, it is empty otherwise. (–) means that the information is not provided. The "Distribution" presents the percentage of each category used in all selected proposals. The gray cells are for the sake of readability to separate different categories.

which CPU architectures are supported by these approaches. The next column indicates which framework(s) have been used by these approaches. If no framework was used, we leave it empty. 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 the distribution of these features over the papers that have been selected based on their relevance to the topic, publication years, and

venues, as discussed in Section 1. For instance, the majority of the solutions (48%) are employing fuzzing, while only 5% of them scan the network.

Key observations of our comparison. 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 4, the majority of the proposed approaches have been evaluated on binary code running on traditional CPU architecture (x86-64). Even though many of the approaches claim that their work could be adapted to support multiple CPU architectures, almost no work has been specifically designed with this portability. Second, as mentioned in the previous key observations in relation to Table 3, the majority of reported bugs, vulnerabilities, and zero-day vulnerabilities are discovered by employing fuzzing approaches; however, fuzzing approaches require a significant amount of time. Moreover, these approaches cannot be applied to large-scale and complex software (e.g., Web browsers) or to check a large corpus of software within a reasonable time. Third, using machine learning to generate smart seed improves fuzzing marginally. For example, SmartSeed [103], which utilizes the adversarial neural network to generate smart seeds, can discover the highest number of vulnerabilities and crashes. Finally, combining two or more binary analysis approaches shows promising scalable and accurate results. For instance, integrating fuzzing with symbolic execution or with taint analysis leads to the discovery of more bugs by Driller [139] and other similar approaches (e.g., deepFuzz [21]) inspired by this work.

3.4.3 Discussion. In the following, we discuss the limitations of dynamic analysis and symbolic execution approaches.

Non-trivial benchmarking. 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 using different metrics. There exist DARPA CGC dataset [43], LAVA [52], and LAVA-M datasets [138], however, they are not comprehensive to cover all use cases. Thus, for the sake of evaluation, two large-scale datasets are needed. (i) *Firmware dataset* containing a large number of publicly available firmware images and embedded devices collected from various vendors. (ii) *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. Hence, 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.

Limited code coverage. Code coverage is essential for complete and robust vulnerability discovery. 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).

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 manually by experts [147], using symbolic execution [146], or by a trained machine learning model [103]. Manual test case generation is a long-time process, not scalable, and costs a lot. However, symbolic execution is also not scalable, cannot provide high coverage for complex software, and it is computationally expensive. Furthermore, proposed solutions [139] 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 [103] to generate input samples from provided training samples automatically; however, those solutions have been trained and tested over simple applications, such as simple PDF format or media player files. However, surveyed dynamic analysis approaches [103, 139] have successfully discovered different 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. [98] recommend that an improved 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 inefficiency. The main advantage of the fuzzing approaches is that they can quickly to produce thousands of input cases in a small period of time. However, most of the input cases will be rejected at the early stage of the program execution, since they do not match with the input format. Therefore, applying fuzzing techniques in a traditional way is not that practical. It will take a significant amount of time to find the vulnerabilities, especially those that are residing in the deep locations. Instead, fuzzing approaches require aid from other techniques, such as instrumentation to help provide feedback (e.g., code coverage as with AFL [155]), or taint analysis to identify different input types, such as magic number and length specifier (e.g., as with VUzzer [128]), or symbolic execution to help pass complex conditions (e.g., as with Driller [139]).

Symbolic execution inefficiency. The proposed symbolic execution techniques have shown their usefulness in detecting more vulnerabilities either when they are applied as a stand-alone approach [48] or when they are integrated with other techniques [139]. However, reducing the time complexity of the constraint solvers is still a bottleneck.

4 STATIC ANALYSIS

In this section, we review the static-based code similarity approaches that are applied to embedded system firmware images. To the best of our knowledge, there exist a limited number of works that employ static analysis to find vulnerabilities in normal binaries. For instance, few static analysis approaches [61, 129, 137] are proposed to locate Use-After-Free vulnerabilities [106] and different buffer overflow vulnerabilities. The code similarity problem focuses on statically analyzing two pieces of binary code (e.g., functions) to measure their similarities. These approaches have been used more often in academia, since they offer the perfect use case of comparing code from different architectures. In these techniques, 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. To overcome the scalability issue (C9), some of the existing works propose filtering prior to the comparison. The objective of the filtering process is to exclude highly likely dissimilar functions from the analysis to prune the search space while maintaining the accuracy.

Various static 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

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,

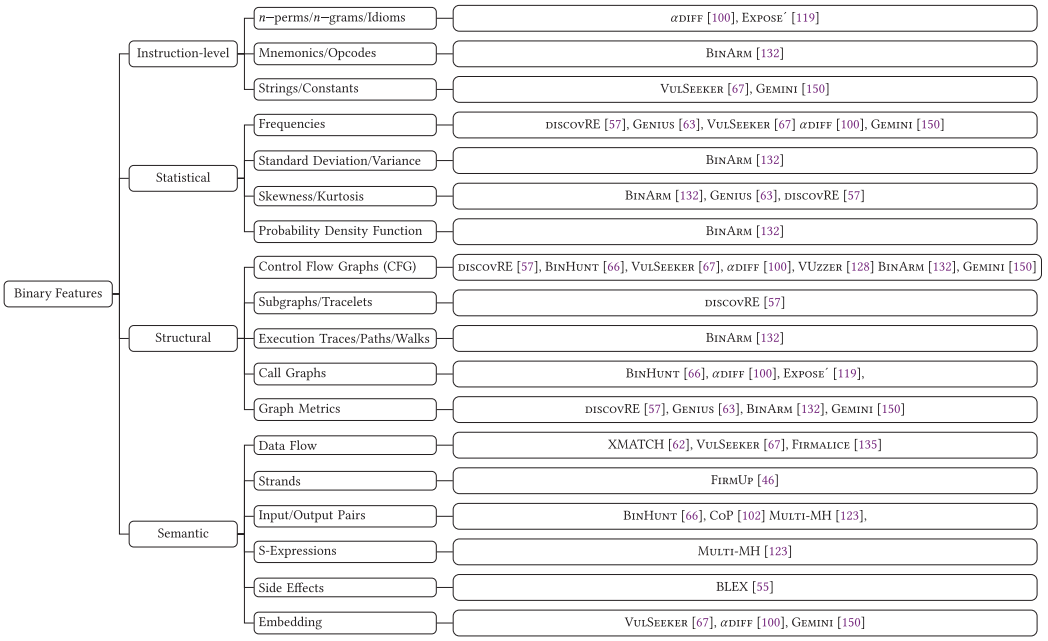

Fig. 3. The proposed taxonomy of the features and corresponding existing approaches.

these features (or a 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.

Instruction-level Features. Instruction-level features can be extracted directly from a given binary code. For instance, *n-grams* [112] 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* [91] 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 [131]. A *mnemonic* of an instruction represents the operations that need to be executed, while typically an *opcode* is the hexadecimal encoding of the instruction [92]. *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.

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 that 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 [95] (e.g., number of arithmetic instruction) is utilized to get more information about the functionality of a function. *Opcode distributions* [7, 18] are used to detect metamorphic malware. Additionally, some statistical distributions such as *skewness* and *kurtosis* [113] and *probability density functions* are computed to get more information about a function and the distribution of its instructions [45, 132, 133].

Structural Features. Structural features represent the semantic information as well as the structural property of a code. *Control flow graphs* (CFGs) [5, 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* [95], which are defined as the short and partial traces of an execution, capture the semantic of execution sequences. Similarly, *execution traces/paths/walks* [6, 27, 132] represent the execution traces while considering basic block semantics. *Call graphs* [64, 127] 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 [72], such as *graph energy* and *betweenness centrality* (node centrality) [116], are used to extract more information about the topology of the graph.

Semantic Features. This category includes features that convey the code semantics to a larger degree. *Execution flow graphs* (EFG) [126] preserve the data and control dependencies between the instructions in a CFG and, therefore, represent the internal structure of a function. *Data dependence* and *program dependence graphs* (PDGs) [107, 134] 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 [81] and value-set analysis [14] 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 [148] at basic block level. *Input/Output pairs* [27, 66, 85, 101, 102, 123] are obtained by the assignment formulas for each basic block, where the effects of input variables on the output variables are monitored to capture their semantics. *S-Expressions* [124] are 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, and so on, that are collected during the program execution and preserve function semantics. *Embeddings* [150, 159] 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 section, 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:

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) [105] distance to additionally consider the similarity between basic blocks, based on several features (e.g., topological order in the function, strings, and constants). Since MCS execution time grows 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 Reference [63], the utilized prefiltering causes notable deduction in accuracy.

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 [118] and further encoded [11] 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 TRENDnet. 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 [132], 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 that 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 [121]. 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.

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, and so on. However, most of the existing data-flow-based solutions cannot be apply at large scale. For instance, MULTI-MH [123] 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[13] [115] to obtain assignment formulas, and then the assignment formulas are simplified to S-Expressions by leveraging $Z3$[14] 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 [85] is used to reduce the complexity of similarity measurement among two basic blocks. Finally, 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 [123]. 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 2,000 firmware images from various device vendors, including NetGear, D-Link, and ASUS.

A semantic-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[15] [51]. The *conditional formulas*

[13] https://github:com/angr/pyvex.
[14] https://github.com/Z3Prover/z3.
[15] https://github.com/trailofbits/mcsema.

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.

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 to find the matching pairs. A cross-architecture binary code similarity approach called Gemini[16] [150] 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 [22] 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[17] [67] extracts the labeled semantic flow graph (LSFG), which combines CFG with DFG, and then proposes a semantics-aware deep neural network model 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 [100], 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 marks the presence of in-depth analysis (e.g., corresponding matched instruction sets or basic blocks) in addition to a final similarity score. The last two columns show which tools are open source and which ones are accessible to the public. Furthermore, in the last row, we provide the distribution (in percentage) of different features over the selected works on static code similarity detection approaches. This section is performed based on their relevance to the topic of the survey (i.e., vulnerability detection in

[16] https://github.com/xiaojunxu/dnn-binary-code-similarity.
[17] https://github.com/buptsseGJ/VulSeeker.

Table 5. A Comparison of State-of-the-art Static-based Code Similarity Detection Approaches

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 [123]	S&P'15		●	●			●		BHB, MinHash	IDA	VEX [115]			
discovRE [57]	NDSS'16			●	●		●		MCS, JD	IDA				
Genius [63]	CCS'16			●	●	●			LSH, JD	IDA				
FirmUp [46]	ASPLOS'18		●				●		DFA, SR	IDA	VEX, LLVM			
Gemini [150]	CCS'17				●			●	DNN	IDA			●	
XMATCH [62]	ASIACCS'17		●	●			●		DFA, GED	IDA	McSema [51]	●		
BinArm [132]	DIMVA'18	●	●	●	●	●			WJD, ED, GM	IDA		●		
VulSeeker [67]	ASE'18		●	●				●	DNN	IDA			●	
αdiff [100]	ASE'18	●		●				●	DNN	IDA				
Distribution	NA	22%	56%	78%	44%	22%	44%	33%	NA	NA	33%	22%	22%	0%

(●) 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 selected proposal. The gray cells are for the sake of readability to separate different categories.

Table 6. A Comparison of State-of-the-art Static-based Code Similarity Detection Implementations and Evaluations

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 [123]	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 [150]	Python	51,314	8,128		●			●	●	●	●	●	●			●	●	2	
XMATCH [62]	-	72	1		●		●	●		●	●	●	●			●	●	4	
BinArm [132]	C++, Python	-	5,756		●				●			●	●		●	●	●	5	●
VulSeeker [67]	Python	-	4,643		●			●	●	●		●	●			●	●	1	
αdiff [100]	-	67,427	2		●		●	●	●	●		●	●			●		6	
Distribution	NA	100%	100%	11%	90%	11%	56%	90%	90%	90%	44%	100%	100%	0%	33%	100%	90%	100%	33%

(●) means the approach provides the corresponding feature, it is empty otherwise. (–) means the information is not provided. "Distribution" presents the percentage of each feature category used in selected proposals. The gray cells are for separating different categories.

firmware images), publication years (January 2004 to January 2020) and venues (e.g., top-tier cybersecurity and software engineering conferences and journals), as discussed in Section 1. For instance, 56% of the surveyed solutions rely on semantic features while only 22% extract instruction-level features.

We further conduct comparative study on the implementation and evaluation of existing works as listed in Table 6. The first and second columns list the proposals and programming languages used in each proposal. The next two columns indicate the type of dataset used for the experiments. The next four columns mark the compilers utilized to prepare the ground truth. In the next three columns, the CPU architectures that are supported by these approaches are marked. In the next two columns, the operating systems is marked. Afterwards, the "Detection" criterion marks the type of detected vulnerable functions (i.e., known/unknown vulnerable function). The next three columns show works that use normalization and provide accuracy and performance results. The next column indicates the number of works that have been compared with the current work. The last column marks whether the work is using any filtering process. We also provide different distributions of features among the surveyed works similarly calculated as in Table 5, such as the ratio of the works that support different CPU architectures. For instance, 90% of the existing 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 that are shown to significantly improve the efficiency and accuracy of the vulnerability detection solutions. 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 that 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. Fourth, 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 that 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 that could not be identified in this manner (C6). Unknown vulnerabilities might be identified by employing data flow and dynamic analysis.

Detecting runtime vulnerabilities. Static approaches fail to detect vulnerabilities that can be exploited during the execution time (C6). For instance, the runtime data-oriented exploits cannot

be detected due to the lack of execution semantics checking [31] or in case of network activities, since this information is provided during the runtime process.

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 static analyses approaches has been somewhat addressed by filtering processes. During this process, the highly likely dissimilar or non-relevant functions will be excluded from the analysis. Therefore, filtration processes minimize the search space 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 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 to validate the bugs and further provide semantic insight on the reason of the execution and the corresponding part of the code. However, 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

Based on our analysis, it is obvious that different embedded systems utilized in a plethora of industry sectors are riddled with vulnerabilities and bugs. Given the massive deployment of such devices, they have become a significant liability due to their attractiveness as vulnerable targets for the attackers.

5.1 Lessons Learned

In the following, we outline some practical lessons learned that could potentially help security practitioners test embedded devices for known and unknown vulnerabilities.

(1) Start by inspecting the target device assumed to contain well-known vulnerable functions by utilizing code similarity approaches leveraging static analysis. Embedded system firmware developers generally utilize software libraries or packages without checking if they have known vulnerabilities. As suggested in the literature, the time needed to assess the vulnerability of an embedded system could be reduced and its robustness could be augmented by using one of the state-of-the-art code similarity approaches, as explained in Section 4.

(2) Re-host targeted device firmware outside its environment for scalable testing. As outlined in Section 3.1.1, if embedded system firmware could be extracted successfully, it is preferable to extensively test it in a scalable manner outside its environment. It is more scalable to utilize

one of the full-emulation approaches; however, if the firmware requires frequent accesses to its peripherals, partial-emulation or virtualization can be used such that the peripherals requests 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, one of the fuzzing approaches mentioned in Section 3.1.2 could be utilized to test the targeted device over one or several of its supported communication protocols. In this case, the type of the used protocol (stateful or stateless) should be taken into account to consider the correct approach during the fuzzing campaign.

5.2 Future Research

In the following, we outline potential further directions in firmware re-hosting, fuzzing, and hybrid approaches.

Automatic firmware re-hosting. Currently available re-hosting techniques (Section 3.1.1) help with integrating advanced binary analysis approaches, such as gray-box fuzzer or symbolic execution into embedded system testing environment. However, proposed emulation solutions for embedded devices are still in their infancy. Building partial or full emulation for embedded systems is a daunting and complicated task. Moreover, re-hosting still requires manual adjustment to make it work for various CPU architectures. As reported in Reference [111], the approach proposed by Eric et al. [73] is an initial effort in this area, and further research in automatic re-host generation will allow for scalable binary firmware analysis. Automatically capturing/learning the original device inputs/outputs and replicating them artificially (through the use of machine learning algorithm, for instance) could also be leveraged to fulfill the need of having valid inputs/outputs for the emulated components and peripherals.

Embedded system fuzzing. Although re-hosting techniques help with introducing advanced fuzzing techniques into embedded device testing, more advanced fuzzing techniques are not proposed yet for embedded device testing, such as test case generation and instrumentation. These techniques are well-known for their wide test coverage and benefits in discovering vulnerable paths. Moreover, different fuzzing techniques have been proposed and evaluated in traditional computing environment, such as PCs and servers, however, not all of them are applicable to embedded systems [109]. In addition to the limited resources of embedded devices (e.g., CPU and RAM), the majority of the embedded systems OSs do not implement memory management. Thus, in the case of a memory crash, no report is generated. Hence, building an emulator to report memory corruption and to provide feedback on the implemented fuzzing approach will provide a chance to examine the effectiveness of different fuzzing approaches on embedded systems. While emulators (e.g., QEMU) have a relatively high impact on the emulated platform performance, they do not emulate all other connected peripherals easily. As such, they may fail in creating a sufficiently rich environment to test potential vulnerabilities effectively. Therefore, it is highly desirable to have an emulator with common emulated components that can be added or removed to replicate the most likely environment in which the firmware is deployed.

Introducing hybrid approaches into embedded system testing. Although different techniques are proposed for embedded devices testing, introducing a hybrid approach is a challenging task; since each approach is adopted for a specific setup and environment, this makes the integration with other tools difficult. For instance, the targeted embedded device may not support the requirement of the instrumentation or taint analysis tools. Moreover, embedded devices

have limited computation resources (e.g., CPU and RAM), and thus those devices cannot host and run computationally expensive techniques (e.g., symbolic execution and taint analysis) without re-hosting their firmware outside the targeted embedded devices. However, in traditional environment these techniques have been combined more frequently and typically provide better results compared to the case where they are utilized separately. For instance, fuzzing and symbolic execution have been combined with other methods, such as taint analysis or static analysis [60, 66, 75, 139, 146]. Therefore, integrating different approaches together for testing embedded devices is still an open area of research.

Multi-architecture support. The high diversity of microcontroller architectures in the embedded world makes it especially hard to develop a "one-size-fits-all" tool without considerable efforts in contrast to desktop computing platforms, which pretty much all rely on the x86-x64 instruction set, making comparison easier. It is hard to know if a given firmware shares commonalities with another if one is based on a Zilog processor and the other is based on an ARM processor. Both of their underlying firmware could still have been compiled from the same source code. The functionalities of each processor may be reflected in the way they implement the same source (e.g., C++) in radically different ways (e.g., if one supports point float operations using a dedicated peripheral while the other does not). This is a major problem when landing in the embedded world.

Hardware-in-the-loop simulation. Several embedded platforms require more than the access to their peripherals to be tested. Vehicle on-board computers, for instance, often test various readouts from the engine before completing the boot sequence to ensure that all components are functioning properly before firing the system up. If a peripheral operates but returns invalid readings (e.g., flag readout or invalid response to a serial message), it will default to an error state where it remains inoperative. For these reasons, the operation context of a component is just as important as the component itself in some cases. Hardware-in-the-loop simulators may sometimes be the only available options.

6 CONCLUSION

In this article, 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. 2019. A taxonomy of software integrity protection techniques. *Advances in Computers* 112 (2019), 413–486.

[2] Chengwei Ai, Weiyu Dong, and Zicong Gao. 2020. A novel concolic execution approach on embedded device. In *Proceedings of the 4th International Conference on Cryptography, Security and Privacy (CSP'20)*. 47–52.

[3] Vincent Alimi, Sylvain Vernois, and Christophe Rosenberger. 2014. Analysis of embedded applications by evolutionary fuzzing. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS'14)*. 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. https://docplayer.net/53692826-D5-3-report-on-automated-vulnerability-discovery-techniques.html.

[5] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2015. SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code. *Dig. Investig.* 12 (2015), S61–S71.

[6] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2018. FOSSIL: A resilient and efficient system for identifying FOSS functions in malware binaries. *ACM Trans. Priv. Secur.* 21, 2 (2018), 8.
[7] Saed Alrabaee, Paria Shirani, Lingyu Wang, Mourad Debbabi, and Aiman Hanna. 2018. On leveraging coding habits for effective binary authorship attribution. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'18)*. Springer, 26–47.
[8] Rajeev Alur and Parthasarathy Madhusudan. 2009. Adding nesting structure to words. *J. ACM* 56, 3 (2009), 1–43.
[9] Amazon. 2018. Amazon elastic compute cloud. Retrieved from https://aws.amazon.com/ec2/.
[10] 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 *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. 1093–1110.
[11] Relja Arandjelovic and Andrew Zisserman. 2013. All about VLAD. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'13)*. 1578–1585.
[12] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein gan. *Arxiv Preprint Arxiv:1701.07875* (2017).
[13] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed dynamic automated test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)*. ACM, 12–22.
[14] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction (CC'04)*. Springer, 5–23.
[15] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.* 51, 3 (2018), 1–39.
[16] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security'14)*. 845–860.
[17] Clark Barrett, Daniel Kroening, and Thomas Melham. 2014. Problem solving for the 21st century: Efficient solver for satisfiability modulo theories. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering.
[18] Daniel Bilar. 2007. Opcodes as predictor for malware. *Int. J. Electron. Secur. Dig. Forens.* 1, 2 (2007), 156–168.
[19] Hristo Bojinov, Elie Bursztein, Eric Lovett, and Dan Boneh. 2009. Embedded management interfaces: Emerging massive insecurity. *BlackHat USA* 1, 8 (2009), 14.
[20] Boofuzz. 2019. 2Binwalk: firmware analysis tool. Retrieved from https://boofuzz.readthedocs.io/en/latest.
[21] Konstantin Böttinger and Claudia Eckert. 2016. DeepFuzz: Triggering vulnerabilities deeply hidden in binaries. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16)*. Springer, 25–34.
[22] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a "siamese" time delay neural network. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems (NeurIPS)*. 737–744.
[23] Teresa Nicole Brooks. 2018. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. In *Proceedings of the Science and Information (SAI'18) Conference*. Springer, 1083–1102.
[24] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI'08)*. 209–224.
[25] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'12)*. IEEE, 380–394.
[26] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. 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 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*.
[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 *Proceedings of the Network and Distributed System Security Symposium (NDSS'18)*.
[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 Computer Security Applications Conference (ACSAC'17)*. 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 International Symposium on Software Testing and Analysis (ISSTA'07)*. 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 *Proceedings of the 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 Asia Conference on Computer and Communications Security (ASIACCS'16)*. ACM, 437–448.
[38] Ang Cui, Michael Costello, and Salvatore J. Stolfo. 2013. When firmware modifications attack: A case study of embedded exploitation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'13)*.
[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 Computer Security Applications Conference (ACSAC'10)*. ACM, 97–106.
[40] Padraig Cunningham and Sarah Jane Delany. 2007. k-nearest neighbour classifiers. *Mult. Class. Syst.* 34, 8 (2007), 1–17.
[41] Johannes Dahse and Thorsten Holz. 2014. Simulation of built-in PHP features for precise static code analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*. Citeseer.
[42] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *Proceedings of the International Conference on Machine Learning (ICML'16)*. 2702–2711.
[43] DARPA. 2018. Cyber Grand Challenge. Retrieved from http://cybergrandchallenge.com.
[44] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Not.* 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 (PLDI'17)*. 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 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, 392–404.
[47] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *ACM SIGPLAN Not.* 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 *Proceedings of the USENIX Security Symposium (USENIX Security'13)*. 463–478.
[49] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Springer, 337–340.
[50] die.net. 2018. Determine file type. Retrieved from https://linux.die.net/man/1/file.
[51] Artem Dinaburg and Andrew Ruef. 2014. Mcsema: Static translation of X86 instructions to LLVM. In *Proceedings of the ReCon 2014 Conference*.
[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 *Proceedings of the IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 110–121.
[53] Dominic Rath. 2018. OpenOCD. Retrieved from http://openocd.org.
[54] Pavel Dovgalyuk. 2012. Deterministic replay of system's execution with multi-target QEMU simulator for dynamic analysis and reverse debugging. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'12)*. 553–556.
[55] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security'14)*. 303–317.
[56] Andrzej Ehrenfeucht. 1961. An application of games to the completeness problem for formalized theories. *Fund. Math* 49, 13 (1961), 129–141.
[57] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*.
[58] ESET. 2018. Vulnerabilities reached a historic peak in 2017. Retrieved from https://bit.ly/2Mgk4x9.

[59] F-Secure. 2015. Vulnerabilities in Foscam IP cameras enable root and remote control. Retrieved from 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 (SSPREW'16)*. ACM, 2.
[61] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *J. Comput. Virol. Hack. Techn.* 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 ACM Asia Conference on Computer and Communications Security (ASIACCS'17)*. 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 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. ACM, 480–491.
[64] Halvar Flake. 2004. Structural comparison of executable objects. In *Proceedings of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'04)*. Gesellschaft für Informatik eV, 161–174.
[65] András Frank. 2005. On Kuhn's Hungarian method—A tribute from Hungary. *Naval Res. Logist.* 52, 1 (2005), 2–5.
[66] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the International Conference on Information and Communications Security (ICICS'08)*. Springer, 238–255.
[67] 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 (ASE'18)*. ACM, 896–899.
[68] François Gauthier, Thierry Lavoie, and Ettore Merlo. 2013. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proceedings of the 29th Computer Security Applications Conference (ACSAC'13)*. ACM, 209–218.
[69] Patrice Godefroid, Michael Y. Levin, David A. Molnar, et al. 2008. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08)*. 151–166.
[70] 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 (ASE'17)*. IEEE Press, 50–59.
[71] Andy Greenberg. 2017. The Reaper IoT Botnet Has Already Infected a Million Networks. Retrieved from https://bit.ly/2SiYZpJ.
[72] C. Griffin. 2012. Graph Theory: Penn State Math 485 Lecture Notes. Retrieved from http://www. personal.psu.edu/cxg286/Math485.pdf.
[73] 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 *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*. 135–150.
[74] H. Craig. 2019. 2Binwalk: firmware analysis tool. Retrieved from https://github.com/ReFirmLabs/binwalk.
[75] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the USENIX Security Symposium (USENIX Security'13)*. 49–64.
[76] 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 (MSR'11)*. ACM, 63–72.
[77] 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 *Proceedings of the USENIX Security Symposium (USENIX Security'12)*, Vol. 8. 1.
[78] Luigi Auriemma. 2018. Signsrch signature identification tool. http://aluigi.altervista.org/mytoolz.htm.
[79] IT Governance Blog. 2018. 6 reasons why software is becoming more vulnerable to cyber attacks. Retrieved from https://bit.ly/2tJq7nu.
[80] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'12)*. IEEE, 48–62.
[81] Ranjit Jhala and Rupak Majumdar. 2005. Path slicing. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 38–47.
[82] 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 *Proceedings of the 3rd International Conference on Data Science in Cyberspace (DSC'18)*. IEEE, 53–60.

[83] 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 (ICSE'07)*. IEEE Computer Society, 96–105.

[84] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. 2016. Survey on software vulnerability analysis method based on machine learning. In *Proceedings of the IEEE International Conference on Data Science in Cyberspace (DSC'16)*. IEEE, 642–647.

[85] Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. 2012. Binary function clustering using semantic hashes. In *Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA'12)*, Vol. 1. IEEE, 386–391.

[86] Nassima Kamel and Jean-Louis Lanet. 2013. Analysis of HTTP protocol implementation in smart card embedded web server. *Int. J. Inf. Netw. Secur.* 2, 5 (2013), 417.

[87] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* 28, 7 (2002), 654–670.

[88] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded security testing with peripheral device caching and runtime program state approximation. In *Proceedings of the 10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE'16)*.

[89] 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 (ASIACCS'14)*. ACM, 329–340.

[90] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the ACM Workshop on Recurring Malcode (WORM'07)*. ACM, 46–53.

[91] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. 2005. Malware phylogeny generation using permutations of code. *J. Comput. Virol.* 1, 1–2 (2005), 13–23.

[92] 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 (MSR'13)*. IEEE Press, 329–338.

[93] 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 *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 447–462.

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

[95] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Workshop on Recent Advances in Intrusion Detection (RAID'05)*. Springer, 207–226.

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

[97] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. 2015. Fuzzing can packets into automobiles. In *Proceedings of the IEEE 29th International Conference on Advanced Information Networking and Applications (AINA'15)*. IEEE, 817–821.

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

[99] 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 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, 627–637.

[100] 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 (ASE'18)*. ACM, 667–678.

[101] 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 (FSE'14)*. ACM, 389–400.

[102] 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 Trans. Softw. Eng.* 43, 12 (2017), 1157–1177.

[103] 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).

[104] 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 Trans. Softw. Eng.* (2019), 1–1. DOI: 10.1109/TSE.2019.2946563

[105] James J. McGregor. 1982. Backtrack search algorithms and the maximal common subgraph problem. *Softw.: Pract. Exper.* 12, 1 (1982).

[106] Mitre. 2018. CWE-416: Use after free. Retrieved from https://cwe.mitre.org/data/definitions/416.html.
[107] Steven S. Muchnick et al. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann.
[108] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Proceedings of the Workshop on Binary Analysis Research (Colocated with the Network and Distributed System Security Symposium)*, Vol. 18. 1–11.
[109] 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 of the Network and Distributed System Security Symposium (NDSS'18)*.
[110] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. 2011. SMS of death: From analyzing to attacking mobile phones on a large scale. In *Proceedings of the USENIX Security Symposium (USENIX Security'11)*. 99.
[111] Marius Munch. 2019. *Dynamic Binary Firmware Analysis: Challenges and Solutions*. Ph.D. Dissertation. Sorbonne Université.
[112] Ginger Myles and Christian Collberg. 2005. K-gram based software birthmarks. In *Proceedings of the ACM Symposium on Applied Computing (SAC'05)*. ACM, 314–318.
[113] Mary Natrella. 2010. NIST/SEMATECH e-handbook of statistical methods. (2010). Retrieved from http://www.itl.nist.gov/div898/handbook/.
[114] Jose Nazario. 2007. *BlackEnergy DDoS Bot Analysis*. Arbor Networks. http://pds15.egloos.com/pds/201001/01/66/BlackEnergy_DDoS_Bot_Analysis.pdf.
[115] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 89–100.
[116] Mark Newman. 2010. *Networks: An Introduction*. Oxford University Press.
[117] James Newsome and Dawn Song. 2005. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium (NDSS'05)*. Citeseer.
[118] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. 2002. On spectral clustering: Analysis and an algorithm. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems (NIPS'02)*. 849–856.
[119] Beng Heng Ng and Atul Prakash. 2013. Expose: Discovering potential binary code re-use. In *Proceedings of the IEEE 37th Computer Software and Applications Conference (COMPSAC'13)*. IEEE, 492–501.
[120] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. 2016. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. IEEE, 780–785.
[121] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH—A locality sensitive hash. In *Proceedings of the 4th Cybercrime and Trustworthy Computing Workshop (CTC'13)*. IEEE, 7–13.
[122] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-fuzz: Fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18)*. IEEE, 697–710.
[123] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. IEEE, 709–724.
[124] 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 Computer Security Applications Conference (ACSAC'14)*. ACM, 406–415.
[125] Protean Security. 2018. Next Generation Dynamic Analysis with PANDA. Retrieved from https://bit.ly/2ZfXlq8.
[126] Jing Qiu, Xiaohong Su, and Peijun Ma. 2016. Using reduced execution flow graph to identify library functions in binary code. *IEEE Trans. Softw. Eng.* 42, 2 (2016), 187–202.
[127] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. 2015. BinComp: A stratified approach to compiler provenance attribution. *Dig. Investig.* 14 (2015), S146–S155.
[128] 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'17)*.
[129] Sanjay Rawat and Laurent Mounier. 2012. Finding buffer overflow inducing loops in binary executables. In *Proceedings of the IEEE 6th International Conference on Software Security and Reliability (SERE'12)*. IEEE, 177–186.
[130] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium (USENIX Security'14)*. 861–875.
[131] Nathan E. Rosenblum. 2011. *The Provenance Hierarchy of Computer Programs*. Ph.D. Dissertation. University of Wisconsin–Madison.
[132] 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 *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'18)*. Springer, 114–138.

[133] Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2017. BinShape: Scalable and robust binary library function identification using function shape. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Springer, 301–324.

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

[135] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice—Automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*.

[136] 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 *Proceedings of the IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 138–157.

[137] Maksim Shudrak. 2017. WinHeap explorer: Efficient and transparent heap-based bug detection in machine code. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 94–101.

[138] Steelix. 2020. LAVA-M. Retrieved from https://sites.google.com/site/steelix2017/home/lava.

[139] 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 *Proceedings of the Network and Distributed System Security Symposium* (NDSS'16). 1–16.

[140] 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.

[141] Gaith Taha. 2007. *Counterattacking the Packers*. McAfee Avert Labs, Aylesbury, UK.

[142] Randy Torrance and Dick James. 2009. The state-of-the-art in IC reverse engineering. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)*. Springer, 363–381.

[143] Fabian Van Den Broek, Brinio Hond, and Arturo Cedillo Torres. 2014. Security testing of GSM implementations. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS'14)*. Springer, 179–195.

[144] Dong Wang, Xiaosong Zhang, Ting Chen, and Jingwei Li. 2019. Discovering vulnerabilities in COTS IoT devices through blackbox fuzzing web management interface. *Secur. Commun. Netw.* 2019 (2019).

[145] 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 (ICSE'18)*. ACM, 61–64.

[146] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 497–512.

[147] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. 2013. RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *KSII Trans. Internet Inf. Syst.* 7, 8 (2013).

[148] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*. IEEE Press, 439–449.

[149] 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).

[150] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, 363–376.

[151] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Computer Security Applications Conference (ACSAC'12)*. ACM, 359–368.

[152] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. 2019. Poster: Fuzzing IoT firmware via multi-stage message generation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. ACM, 2525–2527.

[153] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, 745–761.

[154] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*.

[155] Michal Zalewski. 2010. American fuzzy lop: A security-oriented fuzzer. Retrieved from http://lcamtuf.coredump.cx/afl/ (2010).

[156] 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 ACM SIGSAC Conference on Computer & Communications Security (CCS'13)*. ACM, 487–498.

[157] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the USENIX Security Symposium (USENIX Security'13)*. 337–352.

[158] 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 *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. 1099–1114.

[159] 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).

Received March 2019; revised October 2020; accepted October 2020