REVIEW

Embedded Fuzzing: a Review of Challenges, Tools, and Solutions

Max Camillo Eisele [1*], Marcello Maugeri [2], Rachna Shriwas [3], Christopher Huth [1] and Giampaolo Bella [2]

Abstract

Fuzzing has become one of the best-established methods to uncover software bugs. Meanwhile, the market of embedded systems, which bind the software execution tightly to the very hardware architecture, has grown at a steady pace, and that pace is expected to become yet more sustained in the near future. Also embedded systems benefit from fuzzing, but the innumerable existing architectures and hardware peripherals complicate the development of general and usable approaches, hence a plethora of tools have recently appeared.

Here comes a stringent need for a systematic review in the area of fuzzing approaches for embedded systems, which we term "embedded fuzzing" for brevity. The inclusion criteria chosen in the present article are semi-objective in their coverage of the most relevant publication venues as well as of our personal judgement. The review rests on a formal definition we develop to represent the realm of embedded fuzzing. It continues by discussing the approaches that satisfy the inclusion criteria, then defines the relevant elements of comparison and groups the approaches upon the basis of how the execution environment is served to the System Under Test (SUT). The resulting evaluation produces a table with 42 entries, which in turn supports discussion suggesting vast room for future research due to the limitations currently noted.

Keywords: embedded systems; dynamic analysis; vulnerability mining; embedded security; software security

1 Introduction

Fuzzing is an increasingly popular technique for software testing, namely for findings bugs that could either represent functional problems and vulnerabilities that could be exploited by a malicious attacker. It uses randomness to generate test data for a target with the goal of triggering faults. Faults indicate bugs and can potentially pose a security vulnerability. Because fuzzing is a dynamic method, it analyzes the software while it is executed. By design, dynamic analysis only allows us to find faults that actually occur during execution. Consequently, it is necessary to exercise as many parts of the code and interleaving of branches as possible.

Since fuzzing with pure random input has a small chance to reach large parts of the code, sophisticated fuzzing tools make use of additional information, such as input structure or code-coverage, to generate inputs. A simple but effective approach is to gather code coverage information during input processing of the SUT and collect inputs that trigger previously unreached code parts. This growing collection of inputs, called corpus, is used continuously to generate further inputs.

*Correspondence: MaxCamillo.Eisele@de.bosch.com
[1]Safety, Security and Privacy, Robert Bosch GmbH, Renningen, Germany
Full list of author information is available at the end of the article

Despite its simple underlying principles, fuzzing has proved as an effective method for system and software testing and is recommended by several industry standards. For example, in the *ISO 26262 - Road vehicles - Functional Safety* [1], fuzzing is advocated as one of the testing methods to ensure robustness. Fuzzing can also be found as a recommendation in the *ISA/IEC 62443-4-1 - Secure product development lifecycle requirement* [2]. The recently released *ISO/SAE 21434 - Road vehicles - Cybersecurity* [3] recommends fuzzing as a testing method, too. Additionally, fuzzing can be used in penetration testing, which is recommend in *ISO/IEC/IEEE 291119 - Software and systems engineering — Software testing* [4], *ISO/IEC 12207 - Systems and software engineering — Software life cycle processes* [5], *ISO 27001 - Information technology - Security techniques* [6], *ISO 22301 - Security and resilience* [7].

Fuzzing user (software) applications perhaps is the best-established use of fuzzing, and there are several consolidated techniques for gathering feedback from a target process. For example, the OSS-Fuzz [8] project revealed over 30,000 bugs in 500 open source projects by using coverage-guided fuzzers such as libFuzzer [9], AFL++ [10], and honggfuzz [11].

Another important, growing area for fuzzing pertains to embedded systems, which are microcontroller-based

devices in conjunction with their dedicated software. Typically developed for specific purposes, embedded systems are used pervasively in modern society, and innumerable examples could be made, including smart meters, pacemakers, and factory robots, to just name a few. The market of embedded computing has been growing constantly and is forecasted to keep this trend in the near future [12]. Notably, embedded systems are key components for the Internet of Things (IoT) and for Cyber Physical Systems (CPSs). Therefore, the motivation for fuzzing embedded systems is remarkable.

A first essential feature of embedded systems is that that their firmware is tightly coupled with the specific hardware, including connected peripherals. For example, the firmware of a smart light bulb or of a central heating control panel are both extremely unlikely to work seamlessly on different hardware. A second essential feature of embedded systems is their inherent diversity, which is reflected on the operating systems, CPU architectures, communication mechanisms, and hardware peripherals adopted. For example, while some embedded systems may run Linux-based operating systems, there are some without any operating system at all. Also, while desktop and server systems mainly rely on a few CPU architectures and Operating Systems, these may vastly vary for embedded systems.

We contend that these two features also form the two essential reasons why fuzzing embedded systems is still an open challenge at present [13]. For example, compiling distinct modules, like libraries, into common user applications and exercise fuzzing on them does not work well to test code portions that interact directly with the hardware; incidentally, because of the diverging compiler and environment, this would not test the exact code that ends up on the actual device. It becomes apparent that a reliable, holistic fuzz testing of embedded systems ought to cover both the firmware code as well as the appropriate environment for that firmware. Moreover, the aforementioned diversity poses the biggest challenge due to the need for the fuzzer to scale up to innumerable variants of hardware and firmware that are often poorly documented.

Therefore, we hypothesize that a golden tool and solution for fuzzing embedded systems (*embedded fuzzing* for short) do not exist yet. To verify this hypothesis, we formulate the following research question: *what are the main features and limitations of current tools for fuzzing embedded systems?* To address this question, the present article makes a systematic review of the state of the art on approaches to embedded fuzzing. Our review rests on a formal description of fuzzing for embedded systems and leverages it to advance a clustering of the reviewed works upon the basis of their underlying mechanisms.

The treatment highlights that emulation-based approaches work well for academic examples but may fail on real-world use cases. By contrast, hardware-based approaches with all their incarnations may yield best results albeit not without limitations. Hybrid approaches seem to bear disadvantages from both worlds. Through the presentation of the whole picture on fuzzing for embedded systems, this article draws features as well as limitations of each reviewed work, ultimately demonstrating what kind of future research is needed and deriving directions on how to pursue it.

Section 2 defines the criteria for a piece of research to be included in our review, and Section 3 introduces our extended model for fuzzing embedded systems. After that, we review related work of hardware-based and emulation-based embedded fuzzing in Section 4 and Section 5, respectively. Abstraction-based approaches are reviewed in Section 6. In Section 7 we evaluate the reviewed work, discuss related work in Section 9 and discuss future trends for embedded fuzzing in Section 8. We conclude the paper in Section 10.

2 Inclusion criteria

The inclusion criteria for published material to be included in the present review are:

- C1 Research papers that are published in the top five venues in the category “Engineering & Computer Science”, sub-category “Computer Security & Cryptography” according to Google Scholar [14].
- C2 Research papers that are published during the five years between 2017 and 2021.
- C3 Research papers that mention “fuzzing” and “firmware” or, alternatively, “fuzzing” and “embedded”.
- C4 Research papers or tools that we feel convey relevant approaches to embedded fuzzing.

The first two criteria are objective, as Scholar offers convenient selection and sorting facilities for research venues. The chosen area of security is the one that we found most relevant to fuzzing in general, considering fuzzing as a technique for unveiling software vulnerabilities that an attacker could exploit. To confirm this, we also tried subcategories “Software Systems” and “Computing Systems” but then none of their papers survived the criterion C4. The five venues arising through the first criterion are:

- V1 ACM Symposium on Computer and Communications Security.
- V2 IEEE Transactions on Information Forensics and Security.
- V3 USENIX Security Symposium.
- V4 IEEE Symposium on Security and Privacy.
- V5 Network and Distributed System Security Symposium.

Also criterion C3 is objective. Scholar offers a convenient search facility over the contents of published papers, we queried it with the string "fuzzing AND (firmware OR embedded)" over each of the five identified venues. However, many papers identified this way were not relevant to our purposes for a variety of reasons, ranging from fuzzing being treated only marginally or being mentioned only in the paper references. Here is where criterion C4 comes into place, indicating that we had to exercise manual scrutiny to further select the very contributions that would convey relevant approaches and tools for embedded fuzzing.

Moreover, we decided to appeal to an additional, purposely subjective, inclusion criterion in order to freely represent our experience through the review. It is apparent that criterion C4 does not deliberately refer to a specific time window or venue, hence applying it in isolation from the previous criteria provides us with the freedom of selection we also wanted to have. Therefore, our resulting inclusion criteria can be represented as a sentence in propositional logic:

$$(\text{C1}\wedge\text{C2}\wedge\text{C3}\wedge\text{C4}) \vee (\text{C4} \wedge\neg(\text{C1}\wedge\text{C2}\wedge\text{C3})).$$

Clearly, this sentence is logically equivalent to C4 because our personal judgement had to be applied to all possible candidates. However, its construction allows us to represent the numbers of papers for the meaningful combinations of criteria and venue as well as the papers that we freely decide to consider. Such numbers, in particular for the two main disjuncts in the sentence, can be found in Table 1. It can be understood why our review features a total of 42 papers.

Table 1 Numbers of papers per criterion and venue

	C1∧C2	C1∧C2∧C3	C1∧C2∧C3∧C4
V1	1400	61	2
V2	1350	12	1
V3	716	79	15
V4	518	38	2
V5	315	29	4

C4 ∧¬(C1∧C2∧C3)
18

3 Background and Notation

In this section, a formal description of embedded fuzzing is proposed to mathematically describe fuzzing as a stochastic process. Thereby, the distinct tasks an embedded fuzzer must fulfil are described in an algorithmic manner. We use the notation introduced by Böhme [15] and transform it to fuzzing *systems*.

Let a system $\mathcal{S}$ be our target that we fuzz. The sample space for system $\mathcal{S}$ is the *input space* $\boldsymbol{\mathcal{D}}$. *Fuzzing* is then a stochastic process $(\boldsymbol{\mathcal{D}}, \mathcal{F}, P)$ of selecting inputs t_i from the input space $\boldsymbol{\mathcal{D}}$. The event space $\mathcal{F}$, or *fuzzing campaign*, is then the collection of all drawn input, i.e.

$$\mathcal{F} = \{t_i | t_i \in \boldsymbol{\mathcal{D}}\}_{i=1}^{N} \tag{1}$$

The probability function P dictates the selection of an input t_i with probability p_i to be part of the fuzzing campaign $\mathcal{F}$. Note that we leave out the often used but poorly specified terms black-box, gray-box, or white-box fuzzing. The degree of *smartness* is modeled by adjusting probability function P, i.e. probability p_i for each drawn test input. A tool that implements the sampling function of $(\boldsymbol{\mathcal{D}}, \mathcal{F}, P)$ is called a *fuzzer*.

The probability function P can depend on observations of the system $\mathcal{S}$. If no observations influence the probability p_i for selecting a new input t_i (all p_i's are equal), the *fuzzing campaign* is a uniform random tester[1].

Sampled inputs t_i are processed by system $\mathcal{S}$ with its configuration $\mathfrak{C}$, as in equation 2. The configuration $\mathfrak{C}$ describes the static environment of the system, including hardware properties.

In contrast to existing formal definitions, we introduce an observing mechanism that can observe system $\mathcal{S}$ in desired, not further specified, dimensions. The observation of the system's behavior on processing input t_i is then described by $O_{t_i} \in \boldsymbol{\mathcal{O}}$ and is obtained by

$$O_{t_i} \xleftarrow{observe} \mathcal{S}_{\mathfrak{C}}(t_i), 1 \leq i \leq N, \tag{2}$$

where $\xleftarrow{observe}$ describes the observations of the system during the execution. This construction allows for example to gather code coverage of a system or to observe whether exceptional states of the system have been reached. It also allows for monitoring emitted physical side-channel data or doing liveness checks of the system after a processed input. Further observations can be execution time or the output of a system. The specific observation space depends on the actual device and observer.

For fuzzing, algorithm 1 is built around equation 2, which is called in line 4, where O_{t_i} is the concrete observation of system $\mathcal{S}_{\mathfrak{C}}$ on processing input t_i.

The algorithm continuously samples inputs $t_i \in \boldsymbol{\mathcal{D}}$ on behalf of the probability function P that are then processed by system S. The observation O_{t_i} is inspected for *unspecified behavior* in function SPECIFIED. For example, the specification can contain maximum execution

[1] Even a non-deterministic black-box fuzzer could have some non-empty observations or some non-uniform probabilities.

Algorithm 1: System fuzzing algorithm

Input: System $\mathcal{S}$ with configuration $\mathfrak{C}$, initial seed corpus $\mathbb{C}$, probability function P
Output: Inputs leading to unspecified behavior $T_\times$

$T_\times = \varnothing$
while $\neg($ TIMEOUT() $\vee$ ABORT() $)$ **do** // fuzzing loop
  Pick $t_i \in \mathcal{D}$ with probability p_i // sample input
  $O_{t_i} \xleftarrow{observe} \mathcal{S}_{\mathfrak{C}}(t_i)$
  if $\neg$ SPECIFIED(O_{t_i}) **then**
    $T_\times = \{t_i\} \cup T_\times$ // preserve input
  end
  $P =$ ADJUST(P, O_{t_i}) // may benefit from O_{t_i}
end

durations or illegal states of the system. If unspecified behavior is discovered, the (hopefully) responsible input t_i is preserved in $T_\times$.

Finally, the probability function P may be adjusted by function ADJUST, based on the new observation O_{t_i}. For example, mutation-based coverage-guided fuzzers implicitly alter their probability function, when a new execution path has been discovered by adding the responsible input to an input corpus. On each iteration, a seed is picked from the input corpus and mutated randomly to generate a new input – so the seeds directly influence the probability space of newly sampled inputs.

Differential Fuzzing [16–18] refers to fuzzing of different programs with respect to differences between the observations O_{t_i}, such as coverage or execution time. With an adaption of algorithm 1, systems can be fuzzed differentially, e.g. to test two implementations of the same algorithm for a deviating behavior.

We model *stateful* fuzzing by allowing t_i to contain multiple inputs, $t_i = \langle t_i^1, t_i^2, \ldots, t_i^m \rangle$. Executing such a sequence on system $\mathcal{S}$ brings it to a state s, which we collect as part of $\mathcal{S}$'s observation O_{t_i}.

Ensemble Fuzzing, as introduced by Chen *et al.* [19], is when multiple fuzzers execute algorithm 1. The main idea is that the different tools synchronize their observations. The same system S can be run with different configurations $\mathfrak{C}$ and $\mathfrak{C}'$. For example, configuration $\mathfrak{C}'$ can have the input validation, such as a checksum, turned off to allow a fuzzer to get deeper into the SUT more quickly. The original configuration $\mathfrak{C}$ is then used to validate inputs from configuration $\mathfrak{C}'$ to reduce false positives.

Fuzzing Harness, or *Fuzz Wrapper*, is an adapter between a fuzzer and a specific target. Applications that process data directly from a file or console input channel can most likely be fuzzed without any adapter in between. For all other cases – a typically lightweight – fuzzing harness is necessary to route input data from the fuzzer to the target's interface.

4 Hardware-based Embedded Fuzzing

The high coherency of software and hardware in embedded systems suggests that fuzz testing is to be performed on the actual device. However, already the observing of the device, i.e. implementing $\xleftarrow{observe}$, poses a challenge. In this section, we present projects that aim to run the target application in its designed hardware environment.

Fan *et al.* [20] ported the popular fuzzer AFL to ARM-based IoT devices. Within their project ARM-AFL they developed a code instrumentation strategy for ARM assembly and implemented a lightweight heap memory corruption detector. The whole fuzzing process runs on the target device itself, leading to a high throughput. In principle, the fuzzing process works exactly like fuzzing on a desktop PC. The target process is observed on crash signals and code-coverage in each O_{t_i}. ARM-AFL requires Linux as operating system and the source code of the target program.

FRIDA [21] is a dynamic code instrumentation toolkit that can hook into arbitrary user processes enabling transparent access of the execution. It can also be controlled remotely, allowing for hooking into Linux, QNX, Android and iOS applications. FRIDA offers to collect code coverage data from the hooked process to enable coverage guided fuzzing. However, the FRIDA server application must be executed on the target device, which can be challenging on closed/commercial devices.

Bogdad and Huber [22] developed HARZER ROLLER – a linker-based instrumentation tool for embedded security testing. They address the problem that embedded firmware often needs closed-source libraries in order to communicate with the hardware, which cannot be instrumented by the compiler. These libraries are usually shipped as an object file and are integrated into the firmware by the linker. To be able to generate call traces, all functions within the object file are renamed and appropriate proxy functions are generated. For detecting stack overflows, a stack canary can be generated by the framework before calling the original function. The authors state that this technique is meant for simple embedded devices with limited debug capabilities. The instrumentation of an object file increases its size up to 150%, which usually makes it impossible to instrument all libraries on memory-limited targets. The framework has been used for fuzzing an ESP8266 using BOOFUZZ [23] as black-box fuzzer.

Oh *et al.* [24] present a simple Dynamic Binary Instrumentation (DBI) method for embedded systems without any dependency on the operating system. They connect the target device with a debugger and insert software breakpoints at manually chosen locations. When a breakpoint is reached, the instrumentation framework is notified, and the breakpoint is removed for further

execution. This method enables to observe manually selected, executed code parts in O_{t_i} and could be used for coverage guided fuzzing of any embedded system, which provides a suitable debugger. According to the measurements of the authors, the overhead of this method is only around 1%. However, the measurements have only been done on one device.

Börsig *et al.* [25] present a method to instrument code for ESP32 microcontrollers, whereby the coverage data is returned to the fuzzer's host via a JTAG connection. For this, the source code must be available and the GCC coverage instrumentation mechanism is used. The input data is sent to the target via the original channel, e.g. WiFi. However, the transfer of the coverage data via the JTAG interface slows down the fuzzing process roughly by a factor of ten.

Tychalas *et al.* [26] investigate security evaluation of Programmable Logic Controllers (PLCs). Although, PLC binaries are not regular programs, the authors show that they can introduce vulnerabilities into systems. To reveal such vulnerabilities, they propose a method to instrument PLC binaries, and enable coverage-guided fuzzing on them.

Song *et al.* [27] presented PERISCOPE to examine communication between devices and drivers over Memory-Mapped IO (MMIO) and Direct Memory Access (DMA). The extension PERIFUZZ allows fuzzing on this hardware-OS boundary. PERISCOPE needs to be compiled directly into the target's kernel. Analysis and fuzzing can then be performed directly on present MMIO and DMA regions. For demonstration, AFL is used, but the actual fuzzer is interchangeable.

Delshadtehrani *et al.* [28] designed the programmable hardware monitor PHMon for debugging, assisting vulnerability detection, and enforcing security policies. A prototype of the hardware monitor has been deployed on an Field Programmable Gate Array (FPGA) in conjunction with a RISC-V processor. It can be used to generate coverage feedback directly from the execution on the hardware. The authors state that coverage-guided fuzzing with PHMon and AFL is 16 times faster, than fuzzing in a full system emulator. However, the hardware monitor module needs to be included on the hardware chip directly, to enable this performance advantage.

Sperl *et al.* [29] present a side-channel approach of gathering code coverage from embedded systems by precisely monitoring the power consumption of the target device during execution. Therefore, an oscilloscope is used to record power traces, which are processed further on a host PC to recognize the different executed basic blocks. The recognition is realized by machine learning classification algorithms. With this technique, they are able to approximate the Control Flow Graph (CFG) with correlation coefficients of up to 0.9. For correct results the setup needs to be calibrated and trained on the actual Device under Test (DUT).

García *et al.* [30] utilized timing and electromagnetic emanation side channels from embedded devices for analyzing implementations of cryptographic algorithms. They use these side channels in a specialized feedback-driven fuzzing algorithm to recover cryptographic private keys.

Chen *et al.* [31] present IoTFuzzer, which aims for fuzzing IoT devices that are controlled by mobile phone applications – in this case Android apps only. It makes use of the fact that accompanying mobile apps of IoT devices are aware of the exact protocol and encryption for controlling the device. The idea is to reuse the mobile app to send correct messages to the target device and thereby enabling protocol-aware fuzzing. For this, the mobile app is initially scanned for functions that consume user input and forward it to the IoT device. These functions are then re-used to send fuzzing messages to the target device. This way, the generation of syntactically and semantically correct fuzzing messages is ensured. Crashes are detected by observing the communication or doing liveness checks.

Redini *et al.* [32] have refined this method in their tool DIANE. In contrast to IoTFuzzer, DIANE tries not to hook into the function that consumes user input first, but the last possible one, before the message is encoded and send to the SUT. Thereby, eventual sanitization of the user input within the mobile application is bypassed and the possible input space is enlarged.

Snipuzz [33], also aims to fuzz test IoT devices with accompanying mobile applications. Unlike IoTFuzzer and DIANE, it additionally analyzes responses from the target device to enable feedback-driven fuzzing. Appropriate message sequences are gathered by reading the public API, when it is available, or from analyzing the communication between the accompanying mobile application and the target device. As an alternative, the accompanying mobile application can also be disassembled, but this mostly requires more effort. Although Snipuzz aims to be lightweight, it requires some manual analysis to gather valid initial seeds and select the right message sequences for fuzzing.

Aafer *et al.* [34] present a technique to perform feedback-driven fuzzing of Android TV boxes based on logging outputs. First, static analysis is applied to extract logging statements within the target's firmware. With taint analysis, the collected logging statements are classified on whether they are related to input validation. This labelled collection of logging statements is then used to train a Convolutional Neural Network (CNN) model, that serves as a classifier for logging outputs. During fuzzing, output logs are analyzed by using the model to detect diverging behavior

of the target and to provide feedback to the fuzzer. In addition, they introduce an external component that detects visual and auditory anomalies by capturing and comparing video and audio signals before and after each fuzzing step. This method generates a coarse-grained feedback, compared to branch code coverage, and is designated for rather talkative devices, that give feedback via logs.

5 Emulation-based Embedded Fuzzing

Emulators offer transparency and control of the emulated subject and enable a precise observation O_{t_i} of internal operations in manifold dimensions. Furthermore, multiple instances of an emulator can be created easily, enabling horizontal scaling of the fuzzing process.

However, there are several challenges of running firmware of embedded devices in an emulator, which are carved out well by Wright *et al.* [35]. Most notable for fuzzing is the fidelity and the effort needed to adapt an emulator to a specific target.

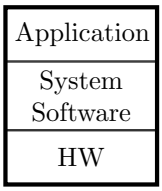

Figure 1 Embedded systems architecture model according to Noergaard [36].

Figure 1 shows an architecture model for embedded systems. While the application logic is contained in the application layer, potential operating systems are located within the system software layer. However, there are embedded systems without a dedicated operating system, often referred to as bare-metal systems. The system software layer then may contain bootloader, drivers, and Hardware Abstraction Layer (HAL) modules. Executing the application within an emulator can be realized by either replacing the hardware layer with a system emulator or move only the application into a user-mode emulator.

In this section, the most notable approaches are presented that enable embedded fuzzing in an emulator.

5.1 User Mode Emulation Fuzzing

User applications that are built for running in an operating system can potentially be executed very easily in an emulator, because of the well-defined operating system interfaces at the application layer. User mode emulation enables fuzzing of binary-only applications with coverage guidance.

It is also possible to transfer user applications from (in particular Linux-based) embedded systems into a user mode emulator like QEMU to perform coverage guided fuzzing, independently from the instruction set architecture. However, accesses to the hardware that embedded applications normally rely on need to be treated adequately by the emulator.

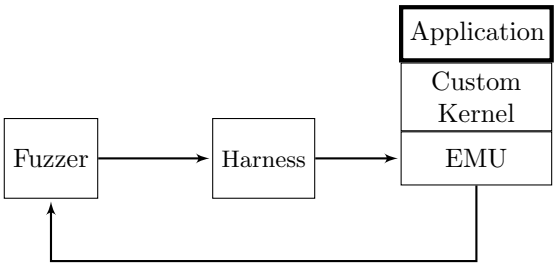

Figure 2 Scheme of Fuzzing Applications in a User-Mode Emulator

All investigated fuzzing frameworks in this category utilize a custom kernel for this purpose, also depicted in Figure 2. The thick boxes depict the parts that originate from the actual target.

Chen *et al.* [37] developed the FIRMADYNE framework, which allows for automated dynamic analysis of Linux-based embedded firmware images. It extracts the root filesystem from a binary firmware image and utilizes a custom kernel to run the image within the QEMU full-system emulator. With this setup, dynamic analysis of the user applications in the firmware can be performed, which is demonstrated by providing a set of known exploits that can be tried on the emulated device. Even though the full-system mode of QEMU is used, FIRMADYNE should be considered to enter at the application layer, because it deploys its own customized kernel and only the user space applications from the firmware are executed. The custom kernel partially compensates for missing hardware emulation, for example by providing an emulated NVRAM that embedded devices often use.

The FIRMADYNE framework is enhanced by Kim *et al.* in FIRMAE [38]. They claim that the FIRMADYNE framework could only get 16.28% of their tested set of firmware images up and running for dynamic analysis. To solve this problem, they introduced heuristics to configure boot parameters, kernel parameters, network interfaces and file systems correctly. With these modifications they were able to automatically run 79.36% of the aforementioned set of firmware images within QEMU.

FIRMFUZZ [39] is an automated introspection and analysis framework for IoT firmware. It aims for embedded devices that offer user interfaces through a webpage

and are based on Linux. The QEMU system emulator is set up with a customized kernel in conjunction with fake peripheral drivers to compensate for potential missing hardware emulation. A headless browser is used to communicate with the device automatically through a virtual network interface to find user interfaces. After the static analysis of the firmware, a generation-based fuzzer is set up. Seed input data is generated, using the contextual information that is gathered from the firmware image. The target is monitored for faults by the modified Linux kernel within the emulator.

FIRM-AFL [40] is based on AFL and FIRMADYNE. The idea is to speed up fuzzing within QEMU by letting the target user process run in the user-mode as long as possible. When necessary, the user process is translated to the full system emulator of the appropriate device hardware. Thereby the overhead of a full system emulation is vastly omitted. The authors state that with this mechanism, the fuzzing process can be sped up by a factor of ten. However, it is required that the target device runs a POSIX-compatible operating system and the hardware can be emulated by QEMU.

Transferring embedded applications from Linux-based devices into an emulator by providing a customized kernel can be successful in some cases, in particular when the target application does not rely on special hardware peripherals. Nevertheless, there remain many embedded systems to which this does not apply, and which demand a different approach for emulation-based fuzzing.

5.2 Full-System Emulation Fuzzing

Once an embedded system can be emulated adequately, code coverage, fault states, and other meta information of the execution can be obtained easily. The next section is about methods that enable the full-system emulation of embedded devices. For a correct emulation of embedded firmware, all hardware peripheral accesses must be treated in the emulator.

5.2.1 Peripheral Emulation

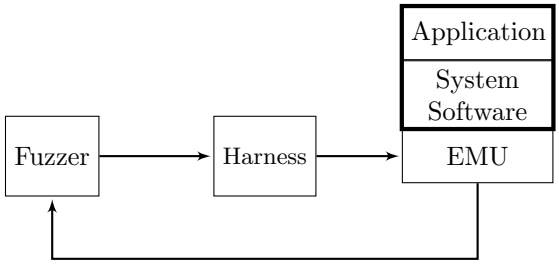

Figure 3 Scheme of Fuzzing Embedded Applications in a Full-System Emulator

A hardware access manifests itself in read and write operations on the hardware address space. Additionally hardware interrupts are a mechanism to let hardware peripherals trigger code areas from the firmware. Implementing software equivalents of hardware peripherals and providing them on their expected locations in the hardware address space is a way to enable emulation. When all peripherals from a target device can be emulated, an unmodified firmware image can be executed and fuzzing can be enabled with little effort as depicted in figure 3.

The QEMU system mode is a popular full-system emulator, which already provides configurations for several microcontrollers and peripherals and supports a large variety of architectures. TRIFORCEAFL [41] combines AFL with QEMU and enables emulation-based coverage-guided fuzzing for targets that can be emulated with QEMU. When the desired target device is not supported, the implementation and configuration can be very laborious and requires deep knowledge of the hardware.

Herdt *et al.* [42] present a different solution for emulating the whole hardware of an embedded system. They apply LIBFUZZER to a SystemC virtual prototype. SystemC is defined as IEEE-1666 standard [43] and provides a set of C++ libraries to define virtual prototypes. Virtual prototypes are models of the entire hardware system and allow an accurate simulation. They are an established way of testing systems during their development in the industry. Fuzzing is performed on the virtual hardware by using a fully booted state of the system, which is preserved by a fork-server mechanism. However the complete system must be described in SystemC, which requires deep insights into the SUT and can cause a lot of manual work again.

Clements *et al.* [44] present HALUCINATOR to address the problem of emulating peripherals by using the HAL as an entry point. First, it locates HAL functions in the firmware through binary analysis. Second, it intercepts the execution of the HAL functions and instead mimics its expected behavior. Handlers for each HAL function must be implemented manually once. Beside correct emulation, HALUCINATOR can intercept functions that provide random values and is able replace them by deterministic functions, which can render fuzzing more efficient.

Kim *et al.* [45] proposed RVFUZZER for detecting *input validation bugs* in robotic vehicles. Robotic vehicles are cyber-physical systems managed in real-time by a microcontroller. It needs to control actuators, process sensor data, and react to control commands. A careful validation of incoming control commands is thereby required, especially if they are received from an unencrypted broadcast medium. RVFUZZER tries to

detect (sequences of) control commands, that bring the robotic vehicle into an unstable state. Therefore, the control program is connected to a physical simulation of the robotic vehicle and input commands as well as environment parameters are mutated. Instabilities are detected by observing whether the presumed state in the control program deviates to much from that in the simulation.

5.2.2 Peripheral Proxying

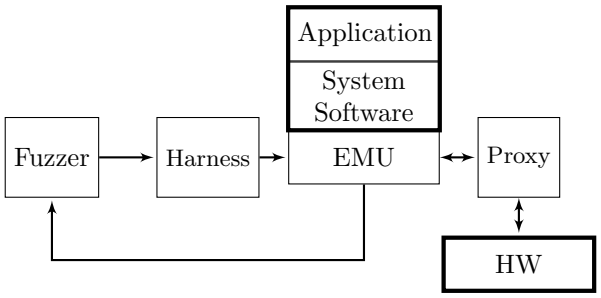

Figure 4 Scheme of Embedded Fuzzing with Peripheral Proxying

When deep knowledge about the SUT is missing, hardware accesses of the firmware must be treated differently. An alternative solution is to forward each hardware access to the real device. Therefore, a proxy application is introduced to route appropriate values and triggered interrupts between the actual hardware and the emulation, as shown in figure 4.

PROSPECT [46] uses TCP/IP connection to forward hardware accesses, AVATAR [47] a debugging connection, and SURROGATES [48] routes hardware accesses through a dedicated FPGA to the actual hardware.

Regarding mobile system drivers, Talebi *et al.* [49] developed CHARM that enables fuzzing device drivers by forwarding hardware peripheral accesses through a USB-based connection. Since the drivers need to be modified for this method, CHARM works with open source drivers only.

AVATAR has a successor, AVATAR^2 [50], which is not only intended for hardware access rerouting, but more for orchestrating different frameworks to enable dynamic analysis. Its flexibility is proven by Muench *et al.* [13].

They enable coverage-guided fuzzing on a wide variety of devices by using PANDA [51] as emulator, AVATAR^2 [50] for forwarding non-emulatable hardware accesses, and BOOFUZZ [23] as fuzzer. Furthermore, they uncover the issue of silent memory corruptions that can occur in embedded devices without Memory Management Units (MMUs) or operating systems that take care of memory accesses. These are memory corruptions that do not result in a crash of the device upon occurrence and are therefore are not easily observable. To detect silent memory corruptions, they present heuristics that can be applied to an emulator, independent from the way of hardware access treatment. When using these heuristics all occurring memory corruptions of a device can be discovered.

Peripheral proxying offers a solution to emulate an embedded device without too much implementation efforts. However, the forwarding of peripheral accesses to the real hardware can depict a bottleneck, depending on the number of requests to the hardware. Additionally, manual configuration and setup of the proxying mechanism is required.

5.2.3 Peripheral Modeling

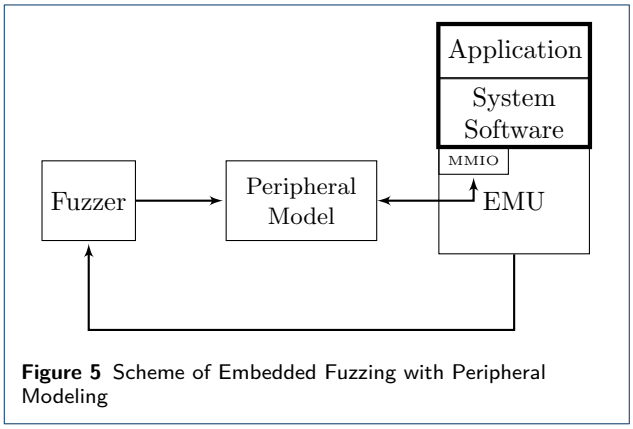

Figure 5 Scheme of Embedded Fuzzing with Peripheral Modeling

When implementing virtual hardware takes too much effort and peripheral proxying is too slow for fuzzing, the automated hardware modeling can be a solution. The idea is to learn how to respond to hardware accesses such that the firmware continues its execution. The peripheral model is thereby directly connected to the MMIO address space and can be supported by the fuzzer, as depicted in figure 5.

Gustafson *et al.* [52] present a semi-automated rehosting framework, called PRETENDER. They solve the modeling of hardware peripherals by preliminary observing and recording of the behavior of the real device with AVATAR^2. Hereby not only accesses to the hardware are recorded, but also the timings and orders of interrupts. Next, a rather complex step of categorizing MMIO registers and initializing *State Approximation model* occurs. This should allow for smart responses to hardware accesses of the firmware. Finally, human interaction is needed to define the entry point of the fuzzing data. The authors state, that PRETENDER allows for a *survivable execution*, which can just be sufficient for a dynamic analysis of the device.

Spensky *et al.* refined this approach with CONWARE [53], which can also learn hardware peripheral behavior by first recording interactions between the firmware and the real hardware peripheral and subsequently extracting models for each of them. The extracted models can then be used for a full system emulation. In contrast to PRETENDER, CONWARE claims to be more generic and even can model peripheral behavior that has not been recorded directly.

Another hardware agnostic approach for embedded fuzzing is presented by Feng *et al.* [54]. Their framework P^{2}IM responds to each peripheral accesses (a read from the MMIO address space) with input data from the fuzzer. Therefore, the MMIO registers are categorized into *Control Registers, Status Registers, Data Registers* and *Control-Status Registers* by observing how the firmware accesses the registers. According to the category, interaction with the registers is treated differently. Most important is the treatment of *Data Registers*, where P^{2}IM directly injects input data from the fuzzer. Thereby, the fuzzer itself models all of the peripheral input generically, omitting the need for finding and choosing the correct input vector for the target. The interrupt emulation is implemented quite pragmatically by sequentially firing one interrupt per 1000 executed basic blocks. When the initially supplied fuzz input buffer is exhausted, the execution is terminated and the code coverage is fed back to the fuzzer. The explorative nature of the fuzzer is used to improve the hardware peripheral modeling successively. The framework supports existing fuzzers to be added as a drop-in component, offering AFL as default. However, peripherals that use DMA are not modeled by P^{2}IM, as this would need insights on the internal design of the target device.

For automatic emulation of DMA input channels in P^{2}IM, Mera *et al.* [55] present the drop-in solution DICE. It observes the behavior of a running firmware in the emulator and recognizes candidates for DMA input channels heuristically. In principle, it searches for pointers to the internal RAM that are written to a memory-mapped IO-registers. The authors claim, that during their tests DICE did not create any false positive categorization and successfully detected 21 out of 22 actively used DMA input channels. With negligible overhead it enables fuzzing of DMA input processing firmwares without further hardware knowledge.

Johnson *et al.* [56] present a more *targeted* peripheral modeling approach with JETSET. Thereby, an analyst manually defines a goal address in the firmware that should be reached and JETSET tries to derive the necessary hardware peripheral responses to reach this address with symbolic execution. For instance, the transition from kernel space to user space can be used as such a goal address. The explicit goal address allows JETSET to mitigate path explosion during symbolic execution.

Zhou *et al.* [57] enable peripheral modeling in their tool µEMU, by mixing symbolic and concrete execution to calculate appropriate responses to a hardware accesses. First, all hardware peripheral dependent inputs are treated symbolically. To avoid path explosion, symbolically calculated values are cached and reused during concrete execution. When invalid execution states are reached, the responsible cached values and the state itself is marked as invalid and different paths are taken by future symbolic executions. This way, the hardware peripherals are enhanced iteratively.

Scharnowski *et al.* [58] refine the mechanism of P^{2}IM. Instead of putting a memory-mapped register into a category, their framework FUZZWARE handles each individual access to a memory-mapped register by additionally considering the program counter on each access. On the first occurrence of an access, the emulator is reset to the instruction right before accessing the memory-mapped register and Dynamic Symbolic Execution (DSE) is used to determine whether and how the value affects the further execution. Accordingly, the individual memory-mapped register access gets just enough random input bits assigned, such that all dependent branches can be reached. This leads to a minimal consumption of input bits from the fuzzer, while fuzzing the whole peripheral interaction. The authors claim, that DMA could also be modeled with further effort as well, but is considered out of scope of their work.

5.3 Sandbox Emulation Fuzzing

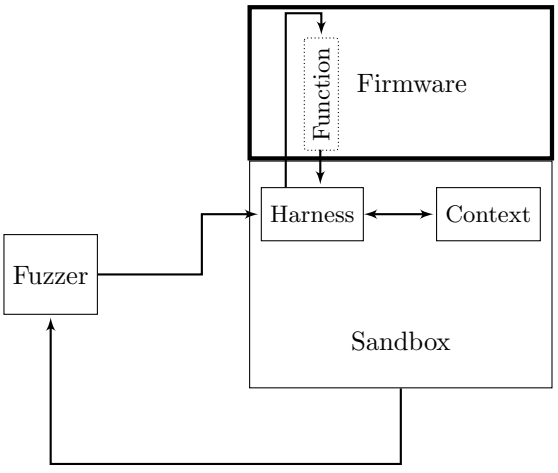

Figure 6 Scheme of Embedded Fuzzing through Sandbox Emulation

In cases, where a full system emulation is not feasible, lightweight sandbox emulation can be a solution. Thereby, the binary code is executed from a manually chosen point with a manually created context. The idea is, to fuzz functions that do not communicate with peripherals at all, thus the hardware peripherals do not need to be emulated. This technique is almost hardware independent, only the instruction set needs to be simulated. Fuzzing a function from a binary firmware file within a sandbox can be realized as shown in figure 6.

MIASM is a reverse engineering tool to analyze, modify and partially emulate binary programs. It offers features like assembling and disassembling for various architectures, emulation with Just-In-Time (JIT) and symbolic execution. In combination with PYTHON-AFL, MIASM can be used to perform fuzzing [59]. Therefore, a sandbox is created by MIASM, input data needs to be mapped to appropriate memory addresses and registers need to be initialized correctly. This technique is mainly interesting for penetration testers, who reverse engineer binaries and want to perform fuzzing of interesting functions in this way. If the source code is available, it is easier to perform fuzzing of hardware independent functions by compiling them into a user application and using a general purpose fuzzer.

The UNICORN CPU Simulator [60] was used by Nathan in [61] in a similar way.

Maier *et al.* present BASESAFE in [62], where they also used the UNICORN CPU SIMULATOR to fuzz different layers of a smartphone baseband chip on manually selected target functions and manually created memory contexts. The downside of these sandbox emulation fuzzing approaches is the constrained and manual selection of the target function and manual creation of the execution context.

A semi-automated approach of supplying an execution context to the target code is presented by Harrison *et al.* [63] with their tool PARTEMU. They present required steps, that allow experts to setup and configure an emulator to enable dynamic analysis of *TrustZones* from embedded systems. Therefore it is explained, when hard and software components should be emulated or reused, and how specific emulation stubs can be implemented. Nevertheless, developing such a emulation based execution context can cause huge manual effort and requires expert knowledge.

Ruge *et al.* present FRANKENSTEIN [64], a highly specialized framework for fuzzing wireless modem firmware in an emulated environment. They run the firmware of a Broadcom Bluetooth chip within QEMU user mode. Through sophisticated reverse engineering, about 100 locations in the code have been determined, where the execution needs to be redirected and substituted manually. This so-called hooking is required to ensure correct emulation of the firmware. With this setup they were able to fuzz the Bluetooth modems of popular mobile phones from Apple and Samsung and unveiled several security problems. However, the setup is highly customized and requires a lot of manual effort to adapt it to other embedded firmware.

An automated sandbox-based fuzzing tool for IoT Firmware is presented by Gui *et al.* with FIRMCORN [65]. First, the firmware image is disassembled and detected functions are rated based on the memory operations they contain and the use of predetermined *sensitive functions*, like `read`, `strcpy`, and `execve`. For high rated functions, a context dump (memory and register values) at the starting point of the function is gathered from the actual device. This allows specific fuzzing of potential vulnerable functions within the CPU emulator UNICORN. An automated mechanism detects crashes of the emulator, which results from missing emulated hardware and skips these crashing functions during further virtual execution. They state that the tool is developed for Linux-based devices only, but it should be possible to extend it for further platforms.

6 Abstraction-based Execution Environment

Symbolic execution is known since several decades [66] and seems not to be located within the domain of fuzzing at a first glance. It analyzes the target program independently from its execution environment. The core idea is to treat all input vectors of a program symbolically (similar to a variable in a mathematical formula) and derive input constraints for all possible program paths. From these constraints, concrete inputs can be extracted that are known to trigger all possible program paths – which is exactly the goal of fuzzing.

However, for each conditional branch in a program, each possible path must be considered in different states. This can lead to the so-called state explosion problem and normally prevents using pure symbolic execution in real-life applications.

6.1 Symbolic Execution of Embedded Firmware

Symbolic execution does not execute the program code directly, but rather interprets it. It is therefore a good candidate to tackle the challenge of lacking hardware peripheral emulation. All values from hardware peripherals can therefore be symbolized and possible program paths can be calculated. However, the more hardware values are symbolized, the more constraints and paths are present (Usually growing exponentially).

Davidson *et al.* [67] implemented FIE, that allows symbolic execution of firmware for MSP430 microcontrollers by using a modified version of KLEE [68].

They assume that software of embedded systems is *simple* enough to allow symbolic execution. Therefore, the target firmware is compiled into a representation that can be symbolically executed with KLEE. FIE includes two notable optimizations: *state pruning* and *memory smudging*. State pruning detects whether the current state has already reached before and prunes it, instead of adding it to the set of active states. The memory smudging function allows us to avoid an intractable state e.g. an infinite loop with an increment inside. In this case the state pruning cannot work because the state is not equivalent due to the presence of the increased variable. The memory smudging sets a threshold for consecutive states that differ only in one memory location.

Corteggiani *et al.* [69] present INCEPTION, a symbolic execution engine for embedded firmwares, also based on the KLEE engine. They added a mechanism to symbolically execute assembly code, which is commonly found in embedded firmware code. Additionally, they enable hardware access forwarding for retrieving concrete values from the actual hardware to reduce the symbolical input space.

6.2 Concolic Execution of Embedded Firmware

Concolic execution refers to the combination of CONCrete and symbOLIC execution. Thereby, traces are used to analyze reached conditions during a concrete execution, and related constraints are derived. +These constraints can be used to generate new input data that exercises a different path of the code. This idea is also termed as hybrid or concolic fuzzing.

Several general purpose hybrid fuzzers, such as QSYM [70], SYMCC [71] are available. Needless to say, that there are also frameworks that focus on concolic execution for embedded firmwares. Herdt *et al.* [72] present an approach to integrate concolic testing engine with SystemC based virtual prototype for the RISK-V architecture. This obviously comes with all the requisites from virtual prototypes again.

Ai *et al.* [73] propose a concolic execution approach for embedded devices that supports various architectures. They perform the concrete execution on the physical device and move the symbolic execution to the host via a debugging connection.

Although concolic execution is a promising method to test code, it faces similar challenges as other embedded fuzzers, because it relies on concrete program traces.

7 Evaluation

Our compact evaluation of the embedded fuzzing approaches reviewed above can be found in table 2. For each reviewed paper, the table columns show what we feel are the relevant elements of comparison. One is whether the fuzzer needs the source code of the SUT to run, and it is clear that this is a major factor for many application scenarios. Another one is whether any prototype tool implementing the proposed approach is readily available and functioning for anyone to use, irrespectively of whether it is open source. Additionally, the table presents the key features as well as the limitations of each candidate approach.

Overall, the wide variety of approaches in the table demonstrates the diversity in the steadily growing research field of embedded fuzzing. Therefore, devising meaningful categories for the existing approaches in order to profitably group the lines in the table requires care and consideration of existing attempts.

Notably, general principles for evaluating and benchmarking traditional fuzzers exist, as proposed by Klees *et al.* [74]. Fuzzers should be tested against a large set of benchmark programs, like GCG [75] or LAVA-M [76] multiple times for at least 24 hours, with the performance plotted over time. The performance should be ideally measured in number of detected bugs. The reached code coverage can be used as a secondary performance measure. Additionally, different sets of seeds should be considered and documented. Arguably, a transfer of these principles to embedded fuzzers would be useful. However, current research on embedded fuzzing still faces more fundamental issues of portability and scalability, namely about enabling a fuzzing approach over the widest possible variety of embedded systems of any complexity.

Wright *et al.* [35] propose to compare different rehosting frameworks in particular on the amount of user interaction needed for the setup, termed as application effort. The application effort refers to the ease of adapting a framework to new targets. Preferably, a framework can be adapted with little knowledge of the target and low configuration effort. It could be measured in the estimation of time needed for the setup, but this would heavily depend on the developer hence be seriously affected by subjectivity.

In light of the existing classification attempts, we feel that the relatively young field of embedded fuzzing may currently be profitably partitioned upon the basis of how the execution environment is served to the SUT. Therefore, we build three essential categories: Hardware-based approaches for those that use the very hardware of the SUT to operate; Emulation-based approaches for those that re-host the firmware of the SUT into an emulator; Abstraction-based approaches for those that abstract away the details of the hardware. We further classify each category according to finer observations.

Hardware-based approaches let the target software execute in its designated environment, therefore we decide to further divide these approaches upon the basis

Table 2 Overview of reviewed embedded fuzzing works

Environment		Framework	Source Code Agnostic	Available	Key Contributions	Limitations
Hardware-based	Instrumentation	ARM-AFL [20]	✗	✗	static instrumentation for ARM code	on-target fuzzing only
		Frida [21]	✓	✓	dynamic instrumentation for various OSes	application on the target required
		Harzer Roller [22]	✓	✗	static instrumentation for object files	only function traces
		Os-less DBI [24]	✗	✗	dynamic instrumentation with breakpoints	manual selection of breakpoint locations
		ESP32 Fuzzing [25]	✗	✓	static instrumentation for ESP32 applications	slow coverage data transmission
		ICSFuzz [26]	✓	✓	static instrumentation for PLC binaries	dedicated to PLCs
		PERIFUZZ [27]	✗	✓	fuzzing at hw-os boundary, driver monitoring	must be compiled into the kernel
		PHMon [28]	✓	✓	hardware module for gathering coverage data	specific hardware required
	Side-Channel	Side-Channel Aware Fuzzing [29]	✓	✗	code-coverage derived from power analysis	calibration needed
		Certified Side Channels [30]	✓	✗	EM and timing side-channels	for crypto libraries only
	Message Interface Reusing	IoTFuzzer [31]	✓	✓	reuse of accompanying mobile applications	not feedback driven, Android-only
		DIANE [32]	✓	✓	enhanced IoTFuzzer mechanism	not feedback driven, Android-only
		Snipuzz [33]	✓	✓	communication analysis for feedback	for unencrypted channels only
		Android TV Fuzzing [34]	✓	✗	using log output for feedback	detailed logs nedded, Android-only
Emulation-based	User Mode Emulation	Firmadyne [37]	✓	✓	custom kernel for emulation	linux-based applications only
		FirmAE [38]	✓	✓	enhanced Firmadyne mechanism	linux-based applications only
		FirmFuzz [39]	✓	✓	fuzzing of IoT configuration webpages	linux-based applications only
		Firm-AFL [40]	✓	✓	speedup by hybrid user and system emulation	linux-based applications only
	Full-System Emulation	TriforceAFL [41]	✓	✓	coverage-guided fuzzing with QEMU	target must be emulatable by QEMU
		SystemC VP Fuzzing [42]	✓	✗	coverage-guided fuzzing on VP	virtual prototype required
		HALucinator [44]	✓	✓	re-hosting at HAL	stubs for HAL libraries required
		RVFuzzer [45]	✓	✗	fuzzing controller for robotic vehicles	rich physical simulation required
	Peripheral Proxying	PROSPECT [46]	✓	✗	peripherals proxying through TCP/IP	requires pthreads and TCP/IP support on target
		SURROGATES [48]	✓	✗	proxying through a custom FPGA	JTAG connection required
		Charm [49]	✗	✓	proxying through USB	recompilation needed
		Avatar^2 [50]	✓	✓	flexible, multi-purpose orchestrating framework	any access to device required
	Peripheral Modeling	PRETENDER [52]	✓	✓	peripheral modeling by recording and learning of peripheral behavior	unseen peripheral behaviour is not modeled
		Conware [53]	✓	✓	additional modeling of unseen peripheral behavior	program for recording must be executed on the target
		P^{2}IM [54]	✓	✓	peripheral modeling by automated classification of requests	missing DMA support
		DICE [55]	✓	✓	modeling of DMA-based peripherals	DMA buffer size not priorly identifiable
		Jetset [56]	✓	✓	peripheral modeling by symbolic execution and manual guidance	manual guidance required
		μEmu [57]	✓	✓	peripheral modeling by concolic execution	caching can cause false hardware modeling
		Fuzzware [58]	✓	✓	peripheral modeling by detailed classification	not for complex systems
	Sandboxing	MIASM [59]	✓	✓	multi-purpose reverse engineering tool	reverse engineering required
		BaseSAFE [62]	✓	✓	coverage-guided fuzzing of baseband chips	manually assembled environment
		PartEMU [63]	✓	✗	coverage-guided fuzzing of *TrustZones*	manually assembled environment
		Frankenstein [64]	✓	✓	coverage-guided fuzzing of wireless fimwares	customized for one specific device
		FIRMCORN [65]	✓	✓	automated sandboxing of functions	linux-based applications only
Abstraction-based	Symbolic Execution	FIE [67]	✗	✓	symbolic execution for MSP430 microcontrollers	complex programs lead to state explosion
		Inception [69]	✗	✓	symbolic execution, even for handwritten assembly and binary libraries	complex programs lead to state explosion
	Concolic Execution	Concolic Testing on VP [72]	✓	✓	Concolic testing of RISC-V virtual prototypes	target must be prototyped
		Concolic Execution on Proxy [73]	✓	✗	symbolic execution on host combined with concrete execution on target	for unix-like systems only

of how they gather feedback from the hardware about the execution of the software. As a result, the hardware category features the three sub-categories Instrumentation, Side-Channel, Message Interface Reusing.

A defining feature for Emulation-based approaches is the way they treat hardware peripheral accesses, so we coherently decide the five sub-categories User Mode Emulation, Full-System Emulation, Peripheral Proxying, Peripheral Modeling and Sandboxing.

The last category features Abstraction-based approaches, hence the two sub-categories for enabling the abstraction process are Symbolic Execution and Concolic Execution. It should be remarked that concolic approaches usually need traces from the execution environment and therefore a concrete execution environment but (manually) selected input vectors can be made symbolic. Therefore, we decide to keep these with Abstraction-based approaches.

8 Discussion

Despite the growing attention and proliferation of embedded systems, the research field of embedded fuzzing still lacks generic solutions. Even comparing different tools remains a big challenge. From our perspective, most tools are evaluated on a small set of targets, chosen by the authors themselves.

The effectiveness of embedded fuzzers can only be evaluated when testing can be done on a large collection of test subjects. A benchmarking suite for embedded fuzzers, could consist of open-source embedded firmwares in conjunction with appropriate hardware peripheral emulation solutions. In this way, different fuzzing strategies could be evaluated on embedded systems instead of relying on the ones that are developed for user applications.

Furthermore, the different characteristics of embedded systems in contrast to user applications should be considered. Traditional fuzzing origins from quickly terminating data processing applications. Embedded systems are rather continuously running systems that usually do not terminate after processing a single input. If the internal state of a system changes during sequences of inputs, it is called stateful. Recently, several fuzzers for stateful software have been proposed [77–80]. Especially, Pham *et al.* [78] have shown that stateful programs, like network servers, have to be fuzzed with awareness of their state to be efficient. Since embedded systems typically are stateful, stateful embedded fuzzing approaches are needed as well.

Most reviewed papers are emulation-based and currently, emulators seem to be the preferred way of enabling embedded fuzzing. Beside their mentioned advantages, there is always the disadvantage of a lower fidelity, which makes it necessary to validate all found bugs on the actual hardware or at least an accurate model of it. This process could be automated by putting the actual device in the loop and testing input candidates immediately.

The other disadvantage of emulators is their setup and configuration effort, to imitate the whole execution environment. However, with the actual hardware, there is an environment already present in which the embedded software runs fine. Therefore, we see more research potential in performing fuzzing on the actual hardware and extract feedback from existing functionalities e.g. debug interfaces. Common embedded debugging tools from *Lauterbach* [81] or *SEGGER* [82] provide real-time tracing mechanisms for a wide variety of microcontrollers, that could be used for fuzzing feedback.

Another, yet rarely handled aspect is that an embedded system does have multiple interfaces, which can be highly entangled. Further research needed to consider the whole system, and not only individual functions, interfaces, or processes while fuzzing. Such a fuzzer could fuzz on multiple interfaces simultaneously, while observing the whole system. Multiple fuzzers or harnesses would need to synchronize their observations, similar to ensemble fuzzing.

Recently, plenty of automated peripheral modeling approaches like P^2IM [54], and FUZZWARE [58] have been proposed. For now, they seem to target rather simple embedded systems. Since they need to model all hardware peripherals that are accessed by the firmware, the approaches do not scale well for more complex systems. Nevertheless, automated peripheral modeling remains one of the most promising method to enable generic embedded fuzzing. Further research in this area could enable emulation-based fuzzing with low application effort for more complex embedded systems, too.

9 Related Work

Detailed summaries of the challenges of fuzzing embedded systems [13] and security analysis of embedded systems [35,83] have been published. However, these reviews do concentrate almost solely on emulation-based approaches. We agree that emulation-based approaches are on the rise but to get the whole picture of embedded fuzzing, hardware-based approaches in all their faces need to be considered, too. We aim to draw such a whole picture and especially want do show the diversity and creativity of the reviewed methods in this paper.

10 Conclusion

In this paper we reviewed the current state of the art of embedded fuzzing. To structure the field, we proposed a formal definition of embedded fuzzing and suggested a taxonomy for it. We carved out the additional challenges of embedded fuzzing compared to the research

field of traditional fuzzing. Furthermore, we have shown that there is currently no easily applicable solution for embedded fuzzing. As traditional fuzzing has already found numerous vulnerabilities in non-embedded software, efficient and easily applicable embedded fuzzing would increase security and integrity of the ubiquitous embedded systems people interact with every day.

Abbreviations

HAL Hardware Abstraction Layer
SUT System Under Test
DBI Dynamic Binary Instrumentation
JIT Just-In-Time
DSE Dynamic Symbolic Execution
CFG Control Flow Graph
DUT Device under Test
IoT Internet of Things
DMA Direct Memory Access
MMIO Memory-Mapped IO
MMU Memory Management Unit
CPS Cyber Physical System
CNN Convolutional Neural Network
PLC Programmable Logic Controller
FPGA Field Programmable Gate Array

Availability of data and materials

All research has been done on publicly available works.

Competing interests

The authors declare that they have no competing interests.

Funding

Not applicable

Authors' contributions

Max Camillo Eisele proposed the categories, ordered all works accordingly, developed the appropriate figures and tables, and wrote many summaries as well as the evaluation, discussion, and conclusion. Marcello Maugeri researched the state of the art on embedded fuzzing and summarized some related tools and works. Rachna Shriwas collected and summarized parts of the related works on embedded fuzzing. Christopher Huth extended the model and algorithm in Section 3 and gave overall guidance for the paper. Giampaolo Bella structured the introduction, defined the inclusion criteria and provided academic advises.

Acknowledgements

We would like to acknowledge all persons who provided feedback and other input to this work.

Author details

[1]Safety, Security and Privacy, Robert Bosch GmbH, Renningen, Germany. [2]Dept. of Math and Computer Science, Università degli Studi di Catania, Catania, Italy. [3]RBEI, Robert Bosch GmbH, Bangalore, India.

References

1. Road vehicles — Functional safety. Standard, International Organization for Standardization, Geneva, CH (December 2018)
2. Secure product development lifecycle requirements. Standard, International Electrotechnical Commission, Geneva, CH (January 2018)
3. Road vehicles — Cybersecurity engineering. Standard, International Organization for Standardization, Geneva, CH (August 2021)
4. Software and systems engineering — Software testing. Standard, International Organization for Standardization, Geneva, CH (September 2013)
5. Systems and software engineering — Software life cycle processes. Standard, International Organization for Standardization, Geneva, CH (November 2017)
6. Information technology — Security techniques — Information security management systems. Standard, International Organization for Standardization, Geneva, CH (October 2013)
7. Security and resilience — Business continuity management systems. Standard, International Organization for Standardization, Geneva, CH (October 2019)
8. Serebryany, K.: Oss-fuzz-google's continuous fuzzing service for open source software (2017)
9. LLVM: libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2021-11-22
10. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: Afl++: Combining incremental steps of fuzzing research. In: 14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20) (2020)
11. Swiecki, R.: honggfuzz - Security oriented software fuzzer. https://honggfuzz.dev/. Accessed: 2021-11-22
12. Alsop, T.: Global Embedded Computing Market Revenue from 2018 to 2027 (in Billion U.S. Dollars). The Insight Partners. Accessed: 2021-03-09 (2019)
13. Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D.: What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In: NDSS (2018)
14. Scholar, G.: Top 20 Computer Security & Cryptography Conferences. https://scholar.google.com/citations?view_op=top_venues&vq=eng_computersecuritycryptography. Accessed: 2021-12-02
15. Böhme, M.: Stads: Software testing as species discovery. ACM Transactions on Software Engineering and Methodology (TOSEM) **27**(2), 1–52 (2018)
16. Nilizadeh, S., Noller, Y., Pasareanu, C.S.: Diffuzz: differential fuzzing for side-channel analysis. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 176–187 (2019). IEEE
17. Noller, Y., Păsăreanu, C.S., Böhme, M., Sun, Y., Nguyen, H.L., Grunske, L.: Hydiff: Hybrid differential software analysis. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 1273–1285 (2020). IEEE
18. He, S., Emmi, M., Ciocarlie, G.: ct-fuzz: Fuzzing for timing leaks. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 466–471 (2020). IEEE
19. Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Jiao, X., Su, Z.: Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 1967–1983 (2019)
20. Fan, R., Pan, J., Huang, S.: Arm-afl: Coverage-guided fuzzing framework for arm-based iot devices. In: International Conference on Applied Cryptography and Network Security, pp. 239–254 (2020). Springer
21. FRIDA Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/. Accessed: 2020-11-04
22. Bogad, K., Huber, M.: Harzer roller: Linker-based instrumentation for enhanced embedded security testing. In: Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium, pp. 1–9 (2019)
23. Pereyda, J.: boofuzz: Network protocol fuzzing for humans. Accessed: Feb **17** (2017)
24. Oh, J., Kim, S., Jeong, E., Moon, S.-M.: Os-less dynamic binary instrumentation for embedded firmware. In: 2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII), pp. 1–3 (2015). IEEE
25. Börsig, M., Nitzsche, S., Eisele, M., Gröll, R., Becker, J., Baumgart, I.: Fuzzing framework for esp32 microcontrollers. In: 2020 IEEE International Workshop on Information Forensics and Security (WIFS), pp. 1–6 (2020). IEEE
26. Tychalas, D., Benkraouda, H., Maniatakos, M.: Icsfuzz: Manipulating i/os and repurposing binary code to enable instrumented fuzzing in {ICS} control applications. In: 30th {USENIX} Security Symposium ({USENIX} Security 21) (2021)
27. Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J.-P., Franz, M.: Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In: NDSS (2019)
28. Delshadtehrani, L., Canakci, S., Zhou, B., Eldridge, S., Joshi, A., Egele, M.: Phmon: a programmable hardware monitor and its security use cases. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 807–824 (2020)
29. Sperl, P., Böttinger, K.: Side-channel aware fuzzing. In: European Symposium on Research in Computer Security, pp. 259–278 (2019).

Springer
30. García, C.P., ul Hassan, S., Tuveri, N., Gridin, I., Aldaya, A.C., Brumley, B.B.: Certified side channels. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 2021–2038 (2020)
31. Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K.: Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In: NDSS (2018)
32. Redini, N., Continella, A., Das, D., De Pasquale, G., Spahn, N., Machiry, A., Bianchi, A., Kruegel, C., Vigna, G.: Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In: 42nd IEEE Symposium on Security and Privacy 2021 (2021)
33. Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., Xiang, Y.: Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. arXiv preprint arXiv:2105.05445 (2021)
34. Aafer, Y., You, W., Sun, Y., Shi, Y., Zhang, X., Yin, H.: Android smarttvs vulnerability discovery via log-guided fuzzing. In: 30th {USENIX} Security Symposium ({USENIX} Security 21) (2021)
35. Wright, C., Moeglein, W.A., Bagchi, S., Kulkarni, M., Clements, A.A.: Challenges in firmware re-hosting, emulation, and analysis. ACM Computing Surveys (CSUR) **54**(1), 1–36 (2021)
36. Noergaard, T.: Embedded systems architecture 2nd edition, a comprehensive guide for engineers and programmers (2012)
37. Chen, D.D., Woo, M., Brumley, D., Egele, M.: Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS, vol. 16, pp. 1–16 (2016)
38. Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., Kim, Y.: Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In: Annual Computer Security Applications Conference 2020 (2020). ACM
39. Srivastava, P., Peng, H., Li, J., Okhravi, H., Shrobe, H., Payer, M.: Firmfuzz: automated iot firmware introspection and analysis. In: Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, pp. 15–21 (2019)
40. Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L.: Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 1099–1114 (2019)
41. Hertz, J., Newsham, T.: TriforceAFL. https://github.com/nccgroup/TriforceAFL. Accessed: 2021-02-09
42. Herdt, V., Große, D., Wloka, J., Güneysu, T., Drechsler, R.: Verification of embedded binaries using coverage-guided fuzzing with systemc-based virtual prototypes. In: Proceedings of the 2020 on Great Lakes Symposium on VLSI, pp. 101–106 (2020)
43. Group, S.-S.C.S.W.: IEEE 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual. https://standards.ieee.org/standard/1666-2011.html (2011)
44. Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M.: Halucinator: Firmware re-hosting through abstraction layer emulation. In: 29th USENIX Security Symposium (USENIX Sec), pp. 1–18 (2020)
45. Kim, T., Kim, C.H., Rhee, J., Fei, F., Tu, Z., Walkup, G., Zhang, X., Deng, X., Xu, D.: Rvfuzzer: finding input validation bugs in robotic vehicles through control-guided testing. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 425–442 (2019)
46. Kammerstetter, M., Platzer, C., Kastner, W.: Prospect: peripheral proxying supported embedded code testing. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp. 329–340 (2014)
47. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., *et al.*: Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: NDSS, vol. 23, pp. 1–16 (2014)
48. Koscher, K., Kohno, T., Molnar, D.: {SURROGATES}: Enabling near-real-time dynamic analyses of embedded systems. In: 9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15) (2015)
49. Talebi, S.M.S., Tavakoli, H., Zhang, H., Zhang, Z., Sani, A.A., Qian, Z.: Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 291–307 (2018)
50. Muench, M., Nisi, D., Francillon, A., Balzarotti, D.: Avatar^2: A multi-target orchestration platform. In: BAR 2018, Workshop on Binary Analysis Research, Colocated with NDSS Symposium, 18 February 2018, San Diego, USA, San Diego, ÉTATS-UNIS (2018).
http://www.eurecom.fr/publication/5437
51. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable reverse engineering with panda. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop. PPREW-5. Association for Computing Machinery, New York, NY, USA (2015). doi:10.1145/2843859.2843867. https://doi.org/10.1145/2843859.2843867
52. Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., *et al.*: Toward the analysis of embedded firmware through automated re-hosting. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019), pp. 135–150 (2019)
53. Spensky, C., Machiry, A., Redini, N., Unger, C., Foster, G., Blasband, E., Okhravi, H., Kruegel, C., Vigna, G.: Conware: Automated modeling of hardware peripherals. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, pp. 95–109 (2021)
54. Feng, B., Mera, A., Lu, L.: P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 1237–1254 (2020)
55. Mera, A., Feng, B., Lu, L., Kirda, E., Robertson, W.: Dice: Automatic emulation of dma input channels for dynamic firmware analysis. arXiv preprint arXiv:2007.01502 (2020)
56. Johnson, E., Bland, M., Zhu, Y., Mason, J., Checkoway, S., Savage, S., Levchenko, K.: Jetset: Targeted firmware rehosting for embedded systems. In: 30th {USENIX} Security Symposium ({USENIX} Security 21) (2021)
57. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic firmware emulation through invalidity-guided knowledge inference. In: 30th {USENIX} Security Symposium ({USENIX} Security 21) (2021)
58. Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., Abbasi, A.: Fuzzware: Using precise mmio modeling for effective firmware fuzzing
59. Guedou: Using Miasm to fuzz binaries with AFL. https://guedou.github.io/talks/2017_BeeRump/slides.pdf (2017)
60. Nguyen, A.Q., Dang, H.V.: Unicorn: Next generation cpu emulator framework. In: Proceedings of the 2015 Blackhat USA Conference (2015)
61. Voss, N.: Fuzzing the Unfuzzable. https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5. Accessed: 2021-02-25
62. Maier, D., Seidel, L., Park, S.: Basesafe: baseband sanitized fuzzing through emulation. In: Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 122–132 (2020)
63. Harrison, L., Vijayakumar, H., Padhye, R., Sen, K., Grace, M.: {PARTEMU}: Enabling dynamic analysis of real-world trustzone software using emulation. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 789–806 (2020)
64. Ruge, J., Classen, J., Gringoli, F., Hollick, M.: Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 19–36 (2020)
65. Gui, Z., Shu, H., Kang, F., Xiong, X.: Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution. IEEE Access **8**, 29826–29841 (2020)
66. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
67. Davidson, D., Moench, B., Ristenpart, T., Jha, S.: FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: 22nd USENIX Security Symposium (USENIX Security 13), pp. 463–478. USENIX Association, Washington, D.C. (2013). https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
68. Cadar, C., Dunbar, D., Engler, D.R., *et al.*: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)
69. Corteggiani, N., Camurati, G., Francillon, A.: Inception: System-wide security testing of real-world embedded systems software. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 309–326 (2018)

70. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (Security 2018) (2018). Distinguished Paper Award Winner. https://www.microsoft.com/en-us/research/publication/qsym-a-practical-concolic-execution-engine-tailored-for-hybrid-fuzzing/
71. Poeplau, S., Francillon, A.: Symbolic execution with symcc: Don't interpret, compile! In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 181–198 (2020)
72. Herdt, V., Große, D., Le, H.M., Drechsler, R.: Early concolic testing of embedded binaries with virtual prototypes: A risc-v case study*. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6 (2019)
73. Ai, C., Dong, W., Gao, Z.: A novel concolic execution approach on embedded device. In: Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy. ICCSP 2020, pp. 47–52. Association for Computing Machinery, New York, NY, USA (2020). doi:10.1145/3377644.3377654. https://doi.org/10.1145/3377644.3377654
74. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138 (2018)
75. 2014 Cyber grand challenge. http://archive.darpa.mil/cybergrandchallenge/about.html. Accessed: 2020-11-13
76. Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R.: Lava: Large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 110–121 (2016). doi:10.1109/SP.2016.15
77. Yu, B., Wang, P., Yue, T., Tang, Y.: Poster: Fuzzing iot firmware via multi-stage message generation. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2525–2527 (2019)
78. Pham, V.-T., Böhme, M., Roychoudhury, A.: Aflnet: a greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 460–465 (2020). IEEE
79. Natella, R.: Stateafl: Greybox fuzzing for stateful network servers. arXiv preprint arXiv:2110.06253 (2021)
80. Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., Holz, T.: Nyx-net: Network fuzzing with incremental snapshots. arXiv preprint arXiv:2111.03013 (2021)
81. Lauterbach: Lauterbach Development Tools. https://www.lauterbach.com. Accessed: 2021-11-22
82. Segger: Segger Debug & Trace Probes. https://www.segger.com/products/debug-trace-probes/. Accessed: 2021-11-22
83. Fasano, A., Ballo, T., Muench, M., Leek, T., Bulekov, A., Dolan-Gavitt, B., Egele, M., Francillon, A., Lu, L., Gregory, N., *et al.*: Sok: Enabling security analyses of embedded systems via rehosting. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, pp. 687–701 (2021)