Cybersecurity

REVIEW Open Access

Embedded fuzzing: a review of challenges, tools, and solutions

Max 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 binds the software execution tightly to the very hardware architecture, has grown at a steady pace, and that pace is anticipated to become yet more sustained in the near future. Embedded systems also 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 this 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 according to how the execution environment is served to the system under test. The resulting review produces a table with 42 entries, which in turn supports discussion suggesting vast room for future research due to the limitations noted.

Keywords: Embedded systems, Dynamic analysis, Vulnerability mining, Embedded security, Software security

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 may 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 of reaching 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.

Despite its simple underlying principles, fuzzing has proved to be an effective method for system and software testing and is recommended by several industry standards. For example, in *ISO 26262—Road vehicles—Functional Safety* (Road Vehicles 2018), fuzzing is advocated as one of the testing methods to ensure robustness. Fuzzing is also found as a recommendation in *ISA/IEC 62443-4-1 - Secure product development lifecycle requirement* (Secure Product Development Lifecycle Requirements 2018). The recently released *ISO/SAE 21434—Road vehicles—Cybersecurity* (Road Vehicles 2021) recommends fuzzing as a testing method, too. Additionally, fuzzing is used in penetration testing, which is recommend in *ISO/IEC/IEEE 291119—Software and systems engineering*

*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

– *Software testing* (Software and Systems Engineering 2013), *ISO/IEC 12207—Systems and software engineering—Software life cycle processes* (Systems and Software Engineering 2017), *ISO 27001—Information technology - Security techniques* (Information Technology 2013), *ISO 22301—Security and resilience* (Security and Resilience 2019).

Fuzzing user (software) applications is perhaps the best-established use of fuzzing, and there are several consolidated techniques for gathering feedback from a target process. For example, the OSS-Fuzz (Serebryany 2017) project revealed over 30,000 bugs in 500 open source projects by using coverage-guided fuzzers such as libFuzzer (LLVM 2021), AFL++ (Fioraldi et al. 2020), and honggfuzz (Swiecki 2021).

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 name just a few. The market of embedded computing has been growing constantly and this trend is expected to continue in the near future (Alsop 2019). 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 in the operating systems, CPU architectures, communication mechanisms, and hardware peripherals adopted. For example, while some embedded systems may run Linux-based operating systems, some run without any operating system at all. Also, while desktop and server systems mainly rely on a few CPU architectures and operating systems, these may vary significantly 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 (Muench et al. 2018). For example, compiling distinct modules, such as libraries, into common user applications and exercising fuzzing on them is not an effective means of testing 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 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, this article conducts a systematic review of the state of the art of 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 taxonomy criteria used to categorize the reviewed works is presented in "Section Taxonomy criteria".

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. By presenting the whole picture of fuzzing for embedded systems, this article demonstrates 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 Inclusion criteria" defines the criteria for a piece of research to be included in our review, and "Section Background and notation" introduces our extended model for fuzzing embedded systems. Thereafter, we review related work of hardware-based and emulation-based embedded fuzzing in "Sections Hardware-based embedded fuzzing" and "Emulation-based embedded fuzzing", respectively. Abstraction-based approaches are reviewed in "Section Abstraction-based execution environment". We review the relevant works for embedded fuzzing in "Section Reviewing embedded fuzzing works", discuss future trends in "Section Discussion and future directions", and related work in 'Section Related work". We conclude the article in "Section Conclusion".

Inclusion criteria

The inclusion criteria for published material to be included in this 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 (Scholar 2021).
- 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 none of the corresponding 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.

Criterion C3 is also objective. Scholar offers a convenient search facility for the contents of published papers. We searched in each of the five identified venues with following search string:

```
fuzzing AND (firmware OR embedded)
```

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 play, 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:

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

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. The selection process is additionally depicted in Fig. 1.

It can be understood why our review features a total of 42 papers.

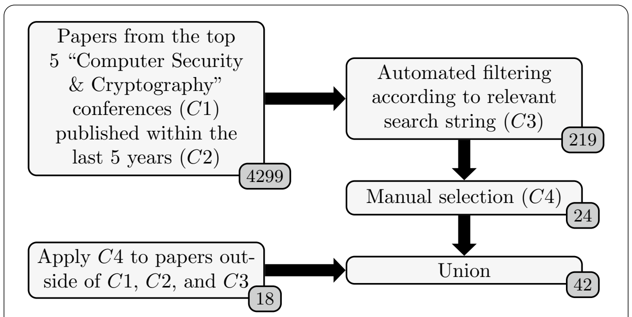
Papers from the top 5 “Computer Security & Cryptography” conferences ($C1$) published within the last 5 years ($C2$)
4299
Automated filtering according to relevant search string ($C3$)
219
Manual selection ($C4$)
24
Apply $C4$ to papers outside of $C1$, $C2$, and $C3$
18
Union
42

Fig. 1 The selection process for finding relevant works, including the numbers of papers each step has mined

Table 1 Numbers of papers per criterion and venue

	C1∧C2	C1∧C2∧C3	C1∧C2∧C3∧ C4	C4 ∧¬ (C1∧C2∧ C3)
V1	1400	61	2	–
V2	1350	12	1	–
V3	716	79	15	–
V4	518	38	2	–
V5	315	29	4	–
Σ	4299	219	24	18

Background and notation

In this section, a formal description of embedded fuzzing is proposed to mathematically describe fuzzing as a stochastic process. Therefore, the distinct tasks an embedded fuzzer must fulfil are described in an algorithmic manner. We use the notation introduced by Böhme (2018) and apply 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 dimensions that are not further specified. The observation of the system's behavior when processing input t_i is then described by $O_{t_i} \in \boldsymbol{\mathcal{O}}$ and is obtained by

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

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 us to monitor emitted physical side-channel data or perform 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.

[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 $T_\times$
1 $T_\times = \varnothing$
2 while $\neg$( Timeout() $\vee$ Abort()) do          // fuzzing loop
3     Pick $t_i \in \boldsymbol{\mathcal{D}}$ with probability $p_i$     // sample input
4     $O_{t_i} \xleftarrow{observe} \mathcal{S}_{\mathfrak{C}}(t_i)$
5     if $\neg$ SPECIFIED($O_{t_i}$) then
6         $T_\times = \{t_i\} \cup T_\times$                 // preserve input
7     end
8     $P$ = ADJUST($P$,$O_{t_i}$)          // may benefit from $O_{t_i}$
9 end
```

The algorithm continuously samples inputs $t_i \in \boldsymbol{\mathcal{D}}$ on behalf of the probability function P, which 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 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 (Nilizadeh et al. 2019; Noller et al. 2020; He 2020) 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. (2019), 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.

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, observing of the device, i.e. implementing $\xleftarrow{observe}$, already poses a challenge. In this section, we present approaches that aim to run the target application in its designed hardware environment.

Fan (2020) ported the popular fuzzer AFL to ARM-based IoT devices. Within their ARM-AFL project 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 the operating system and the source code of the target program.

Frida (FRIDA 2020) is a dynamic code instrumentation toolkit that can hook into arbitrary user processes enabling transparent access to the execution. It can also be controlled remotely, allowing for hooking into Linux, QNX, Android, and iOS applications. In addition, Frida enables the collection of code coverage data from the hooked process to facilitate fuzzing. However, the Frida server application must be executed on the target device, which can be challenging on closed/commercial devices.

Bogad and Huber (2019) 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 (Pereyda 2017) as black-box fuzzer.

Oh et al. (2015) 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 observation of manually selected, executed code parts in O_{t_i} and could be used for coverage-guided fuzzing of any embedded system that 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 performed on one device.

Börsig et al. (2020) 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. (2021) 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. (2019) 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. (2020) 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 a 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 directly on the hardware chip, to enable this performance advantage.

Sperl and Böttinger (2019) 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. (2020) use 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. (2018) 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, thereby enabling protocol-aware fuzzing. For this, the mobile app is initially scanned for functions that consume user input and send it to the IoT device. These functions are then reused 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 performing liveness checks.

Redini et al. (2021) 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 (Feng et al. 2021), 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 usually 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.

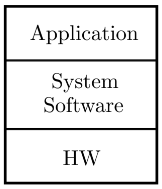

Fig. 2 Embedded systems architecture model according to Noergaard (2012)

Aafer et al. (2021) 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 according to whether they are related to input validation. This labelled collection of logging statements is then used to train a Convolutional Neural Network (CNN) model, which 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.

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, running firmware of embedded devices in an emulator presents several challenges, which are carved out well by Wright et al. (2021). Most notable for fuzzing is the fidelity and the effort needed to adapt an emulator to a specific target.

Figure 2 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 by moving only the application into a user-mode emulator.

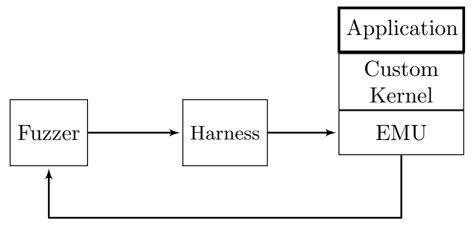

Fig. 3 Scheme of fuzzing applications in a user-mode emulator

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

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 coverag-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.

All investigated fuzzing frameworks in this category use a custom kernel for this purpose, also depicted in Fig. 3. The thick boxes depict the parts that originate from the actual target.

Chen et al. (2016) 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 (Kim et al. 2020). 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 (Srivastava et al. 2019) is an automated introspection and analysis framework for IoT firmware. It is designed 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 (Zheng et al. 2019) 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. As a result, the overhead of a full-system emulation is largely 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.

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 full-system emulation of embedded devices. For a correct emulation of embedded firmware, all hardware peripheral accesses must be treated in the emulator.

Peripheral emulation

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

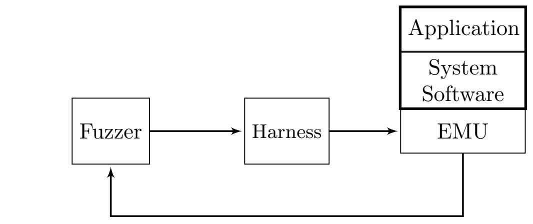

Fig. 4 Scheme of fuzzing embedded applications in a full-system emulator

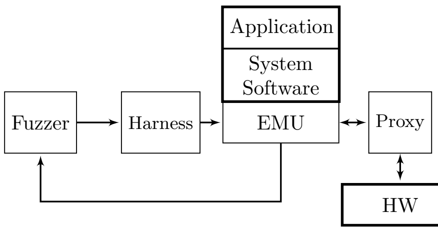

Fig. 5 Scheme of embedded fuzzing with peripheral proxying

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 Fig. 4.

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 (Hertz and Newsham 2021) combines AFL with QEMU and enables emulation-based coverage-guided fuzzing for targets that can be emulated with QEMU. If 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. (2020) 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 (Group S-SCSW 2011) 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 again require a lot of manual work.

Clements et al. (2020) 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. (2019) 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 therefore 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 too much from that in the simulation.

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 Fig. 5.

PROSPECT (Kammerstetter et al. 2014) uses TCP/IP connection to forward hardware accesses, Avatar (Zaddach et al. 2014) a debugging connection, and SURROGATES (Koscher et al. 2015) routes hardware accesses through a dedicated FPGA to the actual hardware.

Regarding mobile system drivers, Talebi et al. (2018) developed Charm that enables fuzzing of device drivers by forwarding hardware peripheral accesses through a USB-based connection. Since the drivers need to be modified for this method, Charm works only with open source drivers.

Avatar has a successor, Avatar^2 (Muench et al. 2018), 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. (2018).

They enable coverage-guided fuzzing on a wide variety of devices by using PANDA (Dolan-Gavitt et al. 2015) as the emulator, Avatar^2 (Muench et al. 2018) for forwarding non-emulatable hardware accesses, and Boofuzz

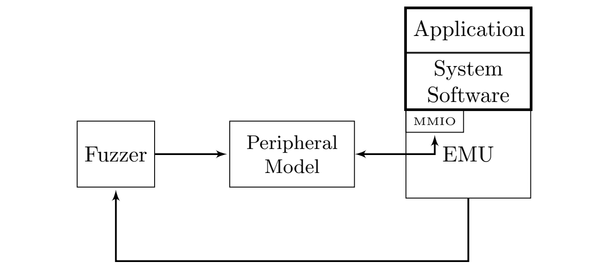

Fig. 6 Scheme of embedded fuzzing with peripheral modeling

(Pereyda 2017) as the 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, regardless of the manner of hardware access treatment. When using these heuristics all, occurring memory corruptions of a device can be discovered.

Peripheral proxying offers a solution for emulating an embedded device without excessive implementation effort. However, the forwarding of peripheral accesses to the real hardware can present a bottleneck, depending on the number of requests to the hardware. Additionally, manual configuration and setup of the proxying mechanism is required.

Peripheral modeling

Where implementing virtual hardware requires too much effort and peripheral proxying is too slow for fuzzing, 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 Fig. 6.

Gustafson et al. (2019) present a semi-automated rehosting framework, called PRETENDER. They solve the modeling of hardware peripherals by means of preliminary observation and recording of the behavior of the real device with AVATAR^2. As a result, 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. (2021) refined this approach with CONWARE, 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 can even model peripheral behavior that has not been recorded directly.

Another hardware-agnostic approach for embedded fuzzing is presented by Feng et al. (2020). Their framework P^2 IM responds to each peripheral access (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. Depending on 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 allows 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 require insights on the internal design of the target device.

For automatic emulation of DMA input channels in P^2 IM, Mera et al. (2020) present the drop-in solution DICE. It observes the behavior of 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 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. (2021) present a more *targeted* peripheral modeling approach with JETSET. In this case, 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. (2021) enable peripheral modeling in their tool μEMU by mixing symbolic and concrete execution to calculate appropriate responses to 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 are marked as invalid and different paths are taken by future symbolic executions. This way, the hardware peripherals are enhanced iteratively.

Scharnowski et al. (2020) 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 is assigned just enough random input bits to ensure 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, but this is considered out of scope of their work.

Sandbox emulation fuzzing

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, meaning that the hardware peripherals do not need to be emulated. This technique is almost hardware-independent since only a simulator for the respective instruction set is required. Fuzzing a function from a binary firmware file within a sandbox can be realized as shown in Fig. 7.

MIASM is a reverse engineering tool to analyze, modify, and partially emulate binary programs. It offers features such as 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 (Guedou 2017). Therefore, a sandbox is created by MIASM, input data

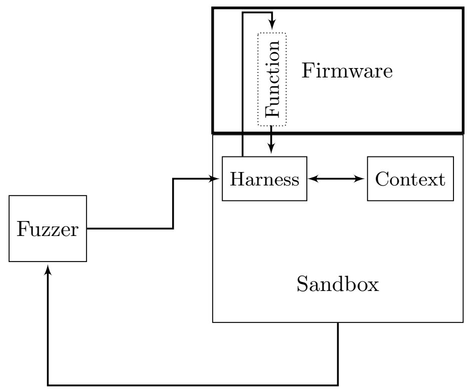

Fig. 7 Scheme of embedded fuzzing through sandbox emulation

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 (Nguyen and Dang 2015) was used by Nathan in Voss (2021) in a similar way.

Maier et al. (2020) present BASESAFE, 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, 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. (2020) with their tool PARTEMU. They present required steps that allow experts to set up and configure an emulator to enable dynamic analysis of *TrustZones* from embedded systems. Therefore, it is explained when hardware and software components should be emulated or reused, and how specific emulation stubs can be implemented. Nevertheless, developing such an emulation-based execution context can involve huge manual effort and requires expert knowledge.

Ruge et al. (2020) present FRANKENSTEIN, 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 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. (2020) with FIRMCORN. 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*, such as `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 result 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 to further platforms.

Abstraction-based execution environment

Symbolic execution is known for several decades (King 1976) and seems not to be located within the domain of fuzzing at 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 (similarly 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 state explosion problem and usually prevents the use of pure symbolic execution in real-life applications.

Symbolic execution of embedded firmware

Symbolic execution does not execute the program code directly, but rather interprets it. It is therefore a good candidate for tackling 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. (2013) implemented FIE, which allows symbolic execution of firmware for MSP430 microcontrollers by using a modified version of KLEE (Cadar et al. 2008). 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 been reached before and prunes it, instead of adding it to the set of active states. The memory smudging function allows 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. (2018) 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.

Concolic execution of embedded firmware

Concolic execution refers to the combination of CONCrete and symbOLIC execution. In this case, 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 (Yun et al. 2018), SymCC (Poeplau and Francillon 2020) are available, as well as frameworks that focus on concolic execution for embedded firmwares. Herdt et al. (2019) present an approach to integrate a concolic testing engine with SystemC-based virtual prototypes for the RISK-V architecture. This is once again subject to all the requirements of virtual prototypes.

Ai et al. (2020) 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.

Reviewing embedded fuzzing works

A summary of the relevant embedded fuzzing works is given in Table 2.

Table 2 Reviewed embedded fuzzing works

Environment		Framework	Source Code Agnostic	Available	Key contributions	Limitations
Hardware-based	Instrumentation	ARM-AFL (Fan 2020)	✗	✗	Static instrumentation for ARM code	On-target fuzzing only
		Frida (FRIDA 2020)	✓	✓	Dynamic instrumentation for various OSes	Application on the target required
		Harzer Roller (Bogad and Huber 2019)	✓	✗	Static instrumentation for object files	Function traces only
		Os-less DBI (Oh et al. 2015)	✗	✗	Dynamic instrumentation with breakpoints	Manual selection of breakpoint locations
		ESP32 Fuzzing (Börsig et al. 2020)	✗	✓	Static instrumentation for ESP32 applications	Slow coverage data transmission
		ICSFuzz (Tychalas et al. 2021)	✓	✓	Static instrumentation for PLC binaries	Dedicated to PLCs
		PERIFUZZ (Song et al. 2019)	✗	✓	Fuzzing at hw-os boundary, driver monitoring	Must be compiled into the kernel
		PHMon (Delshadtehrani et al. 2020)	✓	✓	Hardware module for gathering coverage data	Specific hardware required
	Side-Channel	Side-Channel Aware Fuzzing (Sperl and Böttinger 2019)	✓	✗	Code-coverage derived from power analysis	Calibration needed
		Certified Side Channels (García et al. 2020)	✓	✗	EM and timing side-channels	For crypto libraries only
	Message Interface Reusing	IoTFuzzer (Chen et al. 2018)	✓	✓	Reuse of accompanying mobile applications	Not feedback driven, Android only
		DIANE (Redini et al. 2021)	✓	✓	Enhanced IoTFuzzer mechanism	Not feedback driven, Android only
		Snipuzz (Feng et al. 2021)	✓	✓	Communication analysis for feedback	For unencrypted channels only
		Android TV Fuzzing (Aafer et al. 2021)	✓	✗	Using log output for feedback	Detailed logs needed, Android only
Emulation-based	User Mode Emulation	Firmadyne (Chen et al. 2016)	✓	✓	Custom kernel for emulation	Linux-based applications only
		FirmAE (Kim et al. 2020)	✓	✓	Enhanced Firmadyne mechanism	Linux-based applications only
		FirmFuzz (Srivastava et al. 2019)	✓	✓	Fuzzing of IoT configuration webpages	Linux-based applications only
		Firm-AFL (Zheng et al. 2019)	✓	✓	Speedup by hybrid user and system emulation	Linux-based applications only
	Full-System Emulation	TriforceAFL (Hertz and Newsham 2021)	✓	✓	Coverage-guided fuzzing with QEMU	Target must be emulatable by QEMU
		SystemC VP Fuzzing (Herdt et al. 2020)	✓	✗	Coverage-guided fuzzing on VP	Virtual prototype required
		HALucinator (Clements et al. 2020)	✓	✓	Re-hosting at HAL	Stubs for HALs required
		RVFuzzer (Kim et al. 2019)	✓	✗	Fuzzing controller for robotic vehicles	Rich physical simulation required

Table 2 (continued)

Environment		Framework	Source Code Agnostic	Available	Key contributions	Limitations
	Peripheral Proxying	PROSPECT (Kammerstetter et al. 2014)	✓	✗	Peripherals proxying through TCP/IP	Requires pthreads and TCP/IP support on target
		SURROGATES (Koscher et al. 2015)	✓	✗	Proxying through a custom FPGA	JTAG connection required
		Charm (Talebi et al. 2018)	✗	✓	Proxying through USB	Recompilation needed
		Avatar^2 (Muench et al. 2018)	✓	✓	Flexible, multi-purpose orchestrating framework	Any access to device required
	Peripheral Modeling	PRETENDER (Gustafson et al. 2019)	✓	✓	Peripheral modeling by recording and learning of peripheral behavior	Unseen peripheral behavior is not modeled
		Conware (Spensky et al. 2021)	✓	✓	Additional modeling of unseen peripheral behavior	Program for recording must be executed on the target
		P^{2}IM (Feng et al. 2020)	✓	✓	Peripheral modeling by automated classification of requests	Missing DMA support
		DICE (Mera et al. 2020)	✓	✓	Modeling of DMA-based peripherals	DMA buffer size not identifiable in advance
		Jetset (Johnson et al. 2021)	✓	✓	Peripheral modeling by symbolic execution and manual guidance	Manual guidance required
		µEmu (Zhou et al. 2021)	✓	✓	Peripheral modeling by concolic execution	Caching can cause false hardware modeling
		Fuzzware (Scharnowski et al. 2020)	✓	✓	Peripheral modeling by detailed classification	Not for complex systems
	Sandboxing	MIASM (Guedou 2017)	✓	✓	Multi-purpose reverse engineering tool	Reverse engineering required
		BaseSAFE (Maier et al. 2020)	✓	✓	Coverage-guided fuzzing of baseband chips	Manually assembled environment
		PartEMU (Harrison et al. 2020)	✓	✗	Coverage-guided fuzzing of *TrustZones*	Manually assembled environment
		Frankenstein (Ruge et al. 2020)	✓	✓	Coverage-guided fuzzing of wireless firmwares	Customized for one specific device
		FIRMCORN (Gui et al. 2020)	✓	✓	Automated sandboxing of functions	Linux-based applications only
Abstraction-based	Symbolic Execution	FIE (Davidson et al. 2013)	✗	✓	Symbolic execution for MSP430 microcontrollers	Complex programs lead to state explosion
		Inception (Corteggiani et al. 2018)	✗	✓	Symbolic execution, even for handwritten assembly and binary libraries	Complex programs lead to state explosion

Table 2 (continued)

Environment		Framework	Source Code Agnostic	Available	Key contributions	Limitations
	Concolic Execution	Concolic Testing on VP (Herdt et al. 2019)	✓	✓	Concolic testing of RISC-V virtual prototypes	Target must be prototyped
		Concolic Execution on Proxy (Ai et al. 2020)	✓	✗	Symbolic execution on host combined with concrete execution on target	For unix-like systems only

Taxonomy criteria

This section summarizes the criteria used to cluster the relevant embedded fuzzing works.

The *columns* in Table 2 show what we feel are the relevant elements of comparison for each work.

- *Source Code Agnostic*—This criterion indicates whether the fuzzer needs the source code of the SUT to run, which is a major factor for many application scenarios.
- *Available*—This criterion indicates whether any implemented tool of the proposed approach is readily available and functioning, irrespective of whether it is open or closed source.
- *Key Contributions & Limitations*—This column presents the key features as well as the limitations of each approach.

The *rows* in Table 2 categorize the works based on the execution environment. The categories are as follows.

- *Hardware-based*
  - Instrumentation
  - Side-Channel
  - Message Interface Reusing
- *Emulation-based*
  - User Mode Emulation
  - Full-System Emulation
  - Peripheral Proxying
  - Peripheral Modeling
  - Sandboxing
- *Abstraction-based*
  - Symbolic Execution
  - Concolic Execution

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

Notably, general principles for evaluating and benchmarking traditional fuzzers exist, as proposed by Klees et al. (2018). Fuzzers should be tested against a large set of benchmark programs, such as GCG (Cyber grand challenge 2014) or LAVA-M (Dolan-Gavitt et al. 2016) multiple times for at least 24 hours, with the performance plotted over time. The performance should ideally be measured in the 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. (2021) propose to compare different re-hosting frameworks particularly with regard to 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, thus making the results highly subjective.

In light of the existing classification attempts, we feel that the relatively young field of embedded fuzzing may currently be partitioned most beneficially on 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 run in its designated environment. Therefore, we decide to further divide these approaches upon the basis of how they gather feedback from the hardware about the execution of the software. Thus the hardware category features the three sub-categories Instrumentation, Side-Channel, and Message Interface Reusing.

A defining feature for emulation-based approaches is the way they treat hardware peripheral accesses. Therefore, 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 noted 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.

Discussion and future directions

Desktop user programs communicate via well defined syscalls and do run in their particular virtual address space. Therefore, fuzzing such programs can benefit from different flavours of feedback and sanitizing options. Similarly, well defined target constraints and boundaries are present for hardware fuzzing. Hardware designs are usually represented in HDLs, where hardware fuzzing approaches can be based on Trippel et al. (2021), Laeufer et al. (2018). In between, embedded fuzzing faces a much less precisely specified environment. Generalized statements about interfaces, the environment, and other circumstances can not be made for embedded applications. In fact, an embedded program is an accumulation of machine code instructions that only function properly together with their intended environment and made assumptions.

This is why 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. It would seem that most tools are evaluated on a small set of targets, chosen by the authors themselves, whereas it would be useful to devise public, independent benchmarks.

The effectiveness of embedded fuzzers can only be evaluated when testing can be performed on a large collection of test subjects. A benchmarking suite for embedded fuzzers may consist of open-source embedded firmwares in conjunction with appropriate hardware peripheral emulation solutions. In this way, different fuzzing strategies can 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 originates from quickly terminating data processing applications. Embedded systems, on the other hand, are 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 (Yu et al. 2019; Pham et al. 2020; Natella 2021; Schumilo et al. 2021). In particular, Pham et al. (2020) showed 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 emulators currently 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 may be automated by putting the actual device in the loop and testing input candidates directly.

The other disadvantage of emulators is the setup and configuration effort required to imitate the whole execution environment. However, with the actual hardware, there is an environment already present in which the embedded software runs as expected. Therefore, we see more research potential in performing fuzzing on the actual hardware and extracting feedback from existing functionalities e.g. debug interfaces. Common embedded debugging tools from *Lauterbach* (Lauterbach 2021) or *SEGGER* (Segger 2021) provide real-time tracing mechanisms for a wide variety of microcontrollers, which may be used for fuzzing feedback.

Another albeit rarely handled aspect is that an embedded system has multiple interfaces that can be highly entangled. Further research is 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, similarly to ensemble fuzzing.

Recently, plenty of automated peripheral modeling approaches, such as P^2 IM (Feng et al. 2020) and FUZZWARE (Scharnowski et al. 2020), 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 methods to enable generic embedded fuzzing. Further research in this area could also enable emulation-based fuzzing with low application effort for more complex embedded systems. Another option could be to design generic and reusable HALs to ease re-hosting and enable efficient fuzz testing of hardware-related code. Moreover, as highlighted by Boehme et al. (2020) for traditional fuzzing, we also advocate a larger scope for embedded fuzzers, which should identify a range of vulnerabilities, such as information and timing leakages, and not just bugs.

Future research and tools should aim to unite existing techniques in an embedded ensemble fuzzing framework in order to eliminate their current, individual disadvantages. In addition, such a framework should be cross-architecture, state-aware, and compatible with emulated and real devices. Embedded Fuzzing should consider the whole system in all its details.

Related work

Detailed summaries of the challenges of fuzzing embedded systems (Muench et al. 2018) and security analysis of embedded systems (Fasano et al. 2021; Wright et al. 2021) 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 facets need to be considered, too. We aim to draw such a complete picture and particularly want to highlight the diversity and creativity of the reviewed methods in this article.

Conclusion

This article 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 showed that no easily applicable solution for embedded fuzzing exists. As traditional fuzzing has already found numerous vulnerabilities in non-embedded software, efficient and easily applicable embedded fuzzing would increase the 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: Coverage Guided Fuzzer; VP: Virtual Prototype; POC: Proof-Of-Concept; ISS: Instruction Set Simulator; 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; GDB: GNU Debugger; PLC: Programmable Logic Controller; FPGA: Field Programmable Gate Array; RTL: Register Transfer Level; HDL: Hardware Description Language.

Author contributions

MCE proposed the categories, ordered all works accordingly, developed the appropriate figures and tables, and wrote many summaries. MM researched the state of the art of embedded fuzzing and summarized some related tools and works. RS collected and summarized parts of the related works of embedded fuzzing. CH extended the model and algorithm in "Section Background and notation" and gave overall guidance for the article. GB structured the introduction, defined the inclusion criteria, and reshaped the prose. All authors read and approved the final manuscript.

Funding

Not applicable.

Availability of data and materials

All research has been conducted on publicly available works.

Declarations

Competing interests

The authors declare that they have no competing interests.

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.

Received: 21 April 2021 Accepted: 25 March 2022
Published online: 02 September 2022

References

2014 Cyber grand challenge. http://archive.darpa.mil/cybergrandchallenge/about.html. Accessed 13 Nov 2020

Aafer Y, You W, Sun Y, Shi Y, Zhang X, Yin H (2021) Android smarttvs vulnerability discovery via log-guided fuzzing. In: 30th {USENIX} security symposium ({USENIX} Security 21)

Ai C, Dong W, Gao Z (2020) A novel concolic execution approach on embedded device. In: Proceedings of the 2020 4th international conference on cryptography, security and privacy. ICCSP 2020. Association for Computing Machinery, New York, NY, USA, pp 47–52. https://doi.org/10.1145/3377644.3377654

Alsop T (2019) Global Embedded Computing Market Revenue from 2018 to 2027 (in Billion U.S. Dollars) The Insight Partners. Accessed 9 March 2021

Boehme M, Cadar C, Roychoudhury A (2020) Fuzzing: Challenges and reflections. IEEE Software

Bogad K, Huber M (2019) Harzer roller: Linker-based instrumentation for enhanced embedded security testing. In: Proceedings of the 3rd reversing and offensive-oriented trends symposium, pp 1–9

Böhme M (2018) Stads: Software testing as species discovery. ACM Trans Softw Eng Methodol (TOSEM) 27(2):1–52

Börsig M, Nitzsche S, Eisele M, Gröll R, Becker J, Baumgart I (2020) Fuzzing framework for esp32 microcontrollers. In: 2020 IEEE international workshop on information forensics and security (WIFS). IEEE, pp 1–6

Cadar C, Dunbar D, Engler DR et al (2008) Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol 8, pp 209–224

Chen DD, Woo M, Brumley D, Egele M (2016) Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS, vol 16, pp 1–16
Chen J, Diao W, Zhao Q, Zuo C, Lin Z, Wang X, Lau WC, Sun M, Yang R, Zhang K (2018) Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In: NDSS
Chen Y, Jiang Y, Ma F, Liang J, Wang M, Zhou C, Jiao X, Su Z (2019) Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In: 28th {USENIX} security symposium ({USENIX} Security 19), pp 1967–1983
Clements AA, Gustafson E, Scharnowski T, Grosen P, Fritz D, Kruegel C, Vigna G, Bagchi S, Payer M (2020) Halucinator: Firmware re-hosting through abstraction layer emulation. In: 29th USENIX security symposium (USENIX Sec), pp 1–18
Corteggiani N, Camurati G, Francillon A (2018) Inception: System-wide security testing of real-world embedded systems software. In: 27th {USENIX} security symposium ({USENIX} security 18), pp 309–326
Davidson D, Moench B, Ristenpart T, Jha S (2013) 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. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
Delshadtehrani L, Canakci S, Zhou B, Eldridge S, Joshi A, Egele M (2020) Phmon: a programmable hardware monitor and its security use cases. In: 29th {USENIX} security symposium ({USENIX} Security 20), pp 807–824
Dolan-Gavitt B, Hodosh J, Hulin P, Leek T, Whelan R (2015). 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. https://doi.org/10.1145/2843859.2843867
Dolan-Gavitt B, Hulin P, Kirda E, Leek T, Mambretti A, Robertson W, Ulrich F, Whelan, R (2016) Lava: Large-scale automated vulnerability addition. In: 2016 IEEE symposium on security and privacy (SP), pp 110–121. https://doi.org/10.1109/SP.2016.15
Fan R, Pan J, Huang, S (2020) Arm-afl: Coverage-guided fuzzing framework for arm-based IoT devices. In: International conference on applied cryptography and network security. Springer, pp 239–254
Fasano A, Ballo T, Muench M, Leek T, Bulekov A, Dolan-Gavitt B, Egele M, Francillon A, Lu L, Gregory N et al (2021) 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
Feng B, Mera A, Lu L (2020) P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: 29th {USENIX} security symposium ({USENIX} security 20), pp 1237–1254
Feng X, Sun R, Zhu X, Xue M, Wen S, Liu D, Nepal S, Xiang Y (2021) Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. arXiv preprint arXiv:2105.05445
Fioraldi A, Maier D, Eißfeldt H, Heuse M (2020) Afl++: Combining incremental steps of fuzzing research. In: 14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)
FRIDA Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/. Accessed 4 Nov 2020
García CP, ul Hassan S, Tuveri N, Gridin I, Aldaya AC, Brumley BB (2020) Certified side channels. In: 29th {USENIX} security symposium ({USENIX} security 20), pp 2021–2038
Group S-SCSW (2011) IEEE 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual. https://standards.ieee.org/standard/1666-2011.html
Guedou (2017) Using Miasm to fuzz binaries with AFL. https://guedou.github.io/talks/2017_BeeRump/slides.pdf
Gui Z, Shu H, Kang F, Xiong X (2020) Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution. IEEE Access 8:29826–29841
Gustafson E, Muench M, Spensky C, Redini N, Machiry A, Fratantonio Y, Balzarotti D, Francillon A, Choe YR, Kruegel C et al. (2019) Toward the analysis of embedded firmware through automated re-hosting. In: 22nd international symposium on research in attacks, intrusions and defenses ({RAID} 2019), pp 135–150
Harrison L, Vijayakumar H, Padhye R, Sen K, Grace M (2020) {PARTEMU}: Enabling dynamic analysis of real-world trustzone software using emulation. In: 29th {USENIX} security symposium ({USENIX} Security 20), pp 789–806
He S, Emmi M, Ciocarlie G (2020) ct-fuzz: Fuzzing for timing leaks. In: 2020 IEEE 13th international conference on software testing, validation and verification (ICST). IEEE, pp 466–471
Herdt V, Große D, Le HM, Drechsler R (2019) 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
Herdt V, Große D, Wloka J, Güneysu T, Drechsler R (2020) 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
Hertz J, Newsham T (2021) TriforceAFL. https://github.com/nccgroup/TriforceAFL. Accessed 9 Feb 2021
Information technology—Security techniques—Information security management systems. Standard, International Organization for Standardization, Geneva, CH (2013)
Johnson E, Bland M, Zhu Y, Mason J, Checkoway S, Savage S, Levchenko, K (2021) Jetset: Targeted firmware rehosting for embedded systems. In: 30th {USENIX} security symposium ({USENIX} security 21)
Kammerstetter M, Platzer C, Kastner W (2014) Prospect: peripheral proxying supported embedded code testing. In: Proceedings of the 9th ACM symposium on information, computer and communications security, pp 329–340
Kim T, Kim CH, Rhee J, Fei F, Tu Z, Walkup G, Zhang X, Deng X, Xu D (2019) Rvfuzzer: finding input validation bugs in robotic vehicles through control-guided testing. In: 28th {USENIX} security symposium ({USENIX} security 19), pp 425–442
Kim M, Kim D, Kim E, Kim S, Jang Y, Kim Y (2020) Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In: Annual computer security applications conference 2020. ACM
King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394
Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 2123–2138
Koscher K, Kohno T, Molnar D (2015) *SURROGATES*: Enabling near-real-time dynamic analyses of embedded systems. In: 9th {USENIX} workshop on offensive technologies ({WOOT} 15)
Laeufer K, Koenig J, Kim D, Bachrach J, Sen K (2018) Rfuzz: coverage-directed fuzz testing of RTL on FPGAs. In: 2018 IEEE/ACM international conference on computer-aided design (ICCAD). IEEE, pp 1–8
Lauterbach: Lauterbach Development Tools. https://www.lauterbach.com. Accessed 22 Nov 2021
LLVM: libFuzzer–a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed 22 Nov 2021
Maier D, Seidel L, Park S (2020) 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
Mera A, Feng B, Lu L, Kirda E, Robertson W (2020) Dice: Automatic emulation of DMA input channels for dynamic firmware analysis. arXiv preprint arXiv:2007.01502
Muench M, Nisi D, Francillon A, Balzarotti D (2018) Avatar2: 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. http://www.eurecom.fr/publication/5437
Muench M, Stijohann J, Kargl F, Francillon A, Balzarotti D (2018) What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In: NDSS
Natella R (2021) Stateafl: Greybox fuzzing for stateful network servers. arXiv preprint arXiv:2110.06253
Nguyen AQ, Dang HV (2015) Unicorn: Next generation CPU emulator framework. In: Proceedings of the 2015 Blackhat USA conference
Nilizadeh S, Noller Y, Pasareanu CS (2019). Diffuzz: differential fuzzing for side-channel analysis. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 176–187
Noergaard T (2012) Embedded systems architecture 2nd edition, a comprehensive guide for engineers and programmers
Noller Y, Păsăreanu CS, Böhme M, Sun Y, Nguyen HL, Grunske L (2020). Hydiff: Hybrid differential software analysis. In: 2020 IEEE/ACM 42nd international conference on software engineering (ICSE). IEEE, pp 1273–1285
Oh J, Kim S, Jeong E, Moon S-M (2015) Os-less dynamic binary instrumentation for embedded firmware. In: 2015 IEEE symposium in low-power and high-speed chips (COOL CHIPS XVIII). IEEE, pp 1–3
Pereyda J (2017) boofuzz: Network protocol fuzzing for humans. Accessed 17 Feb

Pham V-T, Böhme M, Roychoudhury A (2020) Aflnet: a greybox fuzzer for network protocols. In: 2020 IEEE 13th international conference on software testing, validation and verification (ICST). IEEE, pp 460–465
Poeplau S, Francillon A (2020) Symbolic execution with symcc: Don't interpret, compile! In: 29th {USENIX} security symposium ({USENIX} Security 20), pp 181–198
Redini N, Continella A, Das D, De Pasquale G, Spahn N, Machiry A, Bianchi A, Kruegel C, Vigna, G (2021) Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In: 42nd IEEE symposium on security and privacy 2021
Road vehicles—Cybersecurity engineering. Standard, International Organization for Standardization, Geneva, CH (2021)
Road vehicles—Functional safety. Standard, International Organization for Standardization, Geneva, CH (2018)
Ruge J, Classen J, Gringoli F, Hollick M (2020) Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In: 29th {USENIX} security symposium ({USENIX} Security 20), pp 19–36
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
Google Scholar Top 20 Computer Security & Cryptography Conferences. https://scholar.google.com/citations?view_op=top_venues&vq=eng_computersecuritycryptography. Accessed 2 Dec 2021
Schumilo S, Aschermann C, Jemmett A, Abbasi A, Holz T (2021) Nyx-net: Network fuzzing with incremental snapshots. arXiv preprint arXiv:2111.03013
Secure product development lifecycle requirements. Standard, International Electrotechnical Commission, Geneva, CH (2018)
Security and resilience—Business continuity management systems. Standard, International Organization for Standardization, Geneva, CH (2019)
Segger: Segger Debug & Trace Probes. https://www.segger.com/products/debug-trace-probes/. Accessed 22 Nov 2021
Serebryany K (2017) Oss-fuzz-google's continuous fuzzing service for open source software
Software and systems engineering—Software testing. Standard, International Organization for Standardization, Geneva, CH (2013)
Song D, Hetzelt F, Das D, Spensky C, Na Y, Volckaert S, Vigna G, Kruegel C, Seifert J-P, Franz M (2019) Periscope: an effective probing and fuzzing framework for the hardware-OS boundary. In: NDSS
Spensky C, Machiry A, Redini N, Unger C, Foster G, Blasband E, Okhravi H, Kruegel C, Vigna G (2021) Conware: automated modeling of hardware peripherals. In: Proceedings of the 2021 ACM Asia conference on computer and communications security, pp 95–109
Sperl P, Böttinger K (2019) Side-channel aware fuzzing. In: European symposium on research in computer security. Springer, pp 259–278
Srivastava P, Peng H, Li J, Okhravi H, Shrobe H, Payer M (2019) 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
Swiecki R honggfuzz - Security oriented software fuzzer. https://honggfuzz.dev/. Accessed 22 Nov 2021
Systems and software engineering—Software life cycle processes. Standard, International Organization for Standardization, Geneva, CH (2017)
Talebi SMS, Tavakoli H, Zhang H, Zhang Z, Sani AA, Qian Z (2018) Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: 27th {USENIX} security symposium ({USENIX} security 18), pp 291–307
Trippel T, Shin KG, Chernyakhovsky A, Kelly G, Rizzo D, Hicks M (2021) Fuzzing hardware like software. arXiv preprint arXiv:2102.02308
Tychalas D, Benkraouda H, Maniatakos M (2021) Icsfuzz: Manipulating i/os and repurposing binary code to enable instrumented fuzzing in *ICS* control applications. In: 30th {USENIX} Security Symposium ({USENIX} Security 21)
Voss N, Fuzzing the Unfuzzable. https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5. Accessed 25 Feb 2021
Wright C, Moeglein WA, Bagchi S, Kulkarni M, Clements AA (2021) Challenges in firmware re-hosting, emulation, and analysis. ACM Comput Surv (CSUR) 54(1):1–36
Yun I, Lee S, Xu M, Jang Y, Kim T (2018) Qsym: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX security symposium (security 2018). Distinguished Paper Award Winner. https://www.microsoft.com/en-us/research/publication/qsym-a-practical-concolic-execution-engine-tailored-for-hybrid-fuzzing/
Yu B, Wang P, Yue T, Tang Y (2019) 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
Zaddach J, Bruno L, Francillon A, Balzarotti D et al (2014) Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: NDSS 23, pp 1–16
Zheng Y, Davanian A, Yin H, Song C, Zhu H, Sun L (2019) Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In: 28th {USENIX} security symposium ({USENIX} security 19), pp 1099–1114
Zhou W, Guan L, Liu P, Zhang Y (2021) Automatic firmware emulation through invalidity-guided knowledge inference. In: 30th {USENIX} security symposium ({USENIX} Security 21)

Publisher's Note