Fuzzing the ARM Cortex-M: A Survey

Silje Marie Sørlien, Åse Marie Solnør, Karin Bernsmed, and Martin Gilje Jaatun

Abstract Security testing low-level connected devices such as Internet of Things (IoT) devices is difficult due to their limited memory, communication bandwidth, and processing power. As a result, traditional testing processes become difficult and time-consuming in such cases. In this paper, we survey recent fuzzers suitable for fuzzing devices with the most common type of microprocessor: the ARM Cortex-M.

Key words: Fuzzing, firmware, analysis, cyber security, testing

1 Introduction

Embedded devices are increasingly connected to critical networks and the internet, emphasizing the need for security testing. This becomes especially challenging with low-level connected devices which have limitations in memory, communication bandwidth, and processing power. As a result, traditional testing processes become difficult and time-consuming in such cases.

We intend to discover, compare and discuss some state-of-the art firmware fuzzers for embedded devices using the ARM Cortex-M, to create an overview

Silje Marie Sørlien
IIK, NTNU, Trondheim, Norway

Åse Marie Solnør
IIK, NTNU, Trondheim, Norway

Karin Bernsmed
IIK, NTNU, Trondheim, Norway
SINTEF Digital, Trondheim, Norway

Martin Gilje Jaatun
SINTEF Digital, Trondheim, Norway, e-mail: martin.g.jaatun@sintef.no

of security testing tools for developers. We will discuss the amount of and quality of documentation for each fuzzer, as most firmware fuzzers available today need extensive manual efforts to configure and analyse the output. Sparse documentation increases the threshold for implementing fuzzing in security testing, and quality guidance material should thus be a priority for developers of these tools. This survey is intended to be a contribution to a not-so-large pool of literature reviews on embedded device fuzzing. It will hopefully highlight some aspects of firmware fuzzing as well as presenting some of the newest research in this field.

The rest of the paper is structured as follows. Section 2 explains the survey method, and Section 3 documents the results. Section 4 provides a discussion of the implications, and Section 5 summarises the results and identifies future work.

2 Method

Our literature review intended to identify, evaluate and narrow down suitable firmware fuzzers. We followed a template for literature reviews including six steps: formulating research questions, performing an initial search for literature, narrowing down the results, assessing the results, extracting interesting information, and lastly analysing the results [28].

We used Google Scholar to identify most up-to-date firmware fuzzers. The main reason for choosing Google Scholar over other databases was that the former also covers grey literature such as reports, blog posts and software documentation. Our search string was "firmware AND fuzzing AND arm cortex M", and gave 3 300 results in Google Scholar. After filtering out papers released before 2020, the results were narrowed down to 1 490. To do a rough elimination of papers not relevant to our research, we read the titles and short summaries available on the Google Scholar search results page. The criteria for this selection was that the paper describes a firmware fuzzer that supports fuzzing ARM Cortex-M CPUs. This left us with approximately 60 possible relevant papers, as illustrated in Fig. 1. When we had identified these, we continued to read abstracts, introductions and conclusions. In addition, we used the *snowball method* where we found firmware fuzzer candidates in other fuzzer papers.

We selected a total of 17 fuzzers we initially found relevant for our research, and read through the papers and their GitHub READMEs to differentiate them. To illustrate and explain the differences between the tools, we included modelled overviews of the architectures of the tools. We adapted them from models from their respective papers, to be able to include them in this paper.

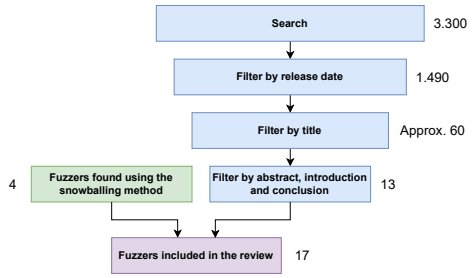

Fig. 1 Search strategy, adapted from [20].

3 Survey Results

Hardware Abstraction Layers (HALs) are abstractions meant to simplify development of firmware implementations. **HALucinator** [4] takes advantage of these to perform High-Level Emulation (HLE), to separate the firmware from the hardware for testing. To be able to do so, a binary analysis is performed to locate library functions of a sample firmware. Following this, generic function implementations can be provided in an emulator for the complete system. They are implemented in an emulator to create an image of the firmware, which can be fuzzed by AFL [4]. The emulator used in HALucinator is based on QEMU [29]. HALucinator supports firmware from ARM Cortex-M architectures that are statically linked into one binary executable. As shown in Fig. 2, HALucinator provides a high-level emulation environment on top of the emulator. The emulator needs to replace the execution of selected function addresses to ensure that the re-hosted firmware is correctly executed. The intercepted functions relates to on-chip or off-chip peripherals of the embedded device. HALucinator simplify this implementation by breaking down the needed implementations per library into *handlers*. These handlers encodes each HAL functions´ semantics. The purpose of *peripheral models* is to address common intrinsic elements of what a certain class or kind of peripheral needs to perform. It can be observed in the figure that the IO server enables emulator host interactions. To be able to meaningfully execute the re-hosted firmware it must be able to interact with external devices outside of the CPU. Because of this, each peripheral model defines an interface for the host system to send and receive data and trigger interrupts. An I/O server is then used to make these interfaces available [4]. The GitHub repository was last updated in November of 2022 [3]. It includes documentation about how to use the fuzzer [3]. It also explains the installation steps, a recommended environment, dependencies and basic usage with examples. Lastly, there is a brief explanation on how to analyse the output.

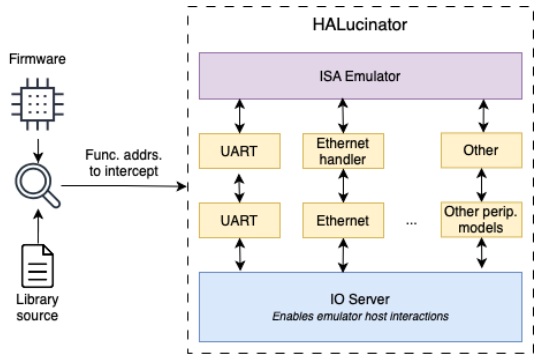

Fig. 2 Overview of HALucinator, adapted from [4].

μAFL [25] is a grey-box fuzzing tool that employs hardware-in-the-loop. This implies that to be able to use μAFL, developers need access to the board as well as additional dongles. Dongles are components that can be plugged into the device to provide more functionality to the device. It leverages debugging tools in existing embedded system development to construct an AFL-compatible environment for fuzzing [25]. It uses the debug dongle to bridge the fuzzing environment on the PC and the target firmware on the microcontroller device. These debug dongles from the embedded development environment are used to ensure cooperability between the engine and AFL [41] [25]. To collect code coverage information without costly code instrumentation, μAFL relies on the ARM Embedded Trace Macrocell (ETM) hardware debugging feature, which transparently collects the instruction trace and streams the results to the PC [25]. The raw ETM data is obscure and needs enormous computing resources to recover the actual instruction trace. μAFL addresses this challenge by using a feedback-driven approach to recover the instruction trace from the raw ETM data. It then uses the recovered instruction trace to guide the fuzzing process. Fig. 3 shows the overview of μAFL. The host PC and target board communicate through a debugging dongle. μAFL adopts AFLs genetic algorithm while substituting its process-based execution engine with two essential components, an online trace collector and an offline trace analyser [25]. The host PC feeds the test case through the debug dongle into a reserved memory on the target board. The target board directs the target to begin execution. The host PC sends the command to activate the ETM function. While the firmware is executing, the generated instruction trace is synchronously streamed to the host PC via the debug dongle. The host PC sends command to deactivate ETM. The collected trace information is used to reconstruct the execution paths. The final outcome is checked against the bitmap to see if any new paths have been found. The Github repository was last updated in July of 2022 [24]. The documentation includes a recommended environment. However, there are no instructions on how to actually install it correctly. It explains how to run the fuzzer and a

provides list on how to prepare the hardware environment. In addition, there is no documentation on how to analyse the output or how to create a correct configuration file.

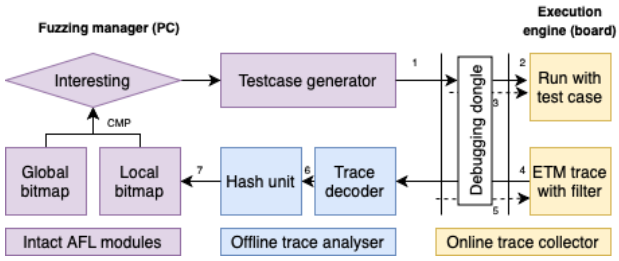

Fig. 3 Overview of μAFL, adapted from [25].

FIRM-AFL [43] is a grey-box fuzzer where the main focus is to create a high-throughput emulation to increase efficiency in full system testing [43]. The paper proposes a new technique called *augmented process emulation*, where the idea is to augment process (user-mode) emulation with full system emulation. FIRM-AFL is a prototype to evaluate the feasibility of this technique, and it is built on top of AFL [41] and FIRMADYNE [1] [43]. FIRMADYNE [1] can emulate both ARM and MIPS architectures [16]. The architecture and workflow of FIRM-AFL is shown in Fig. 4. *afl-fuzz* that is the main program that drives the fuzzing. This program picks a seed from the seed queue, performs a random mutation, generates an input, and feeds this input to the target program. To gather code coverage data during the target program's execution, FIRM-AFL uses augmented process emulation to launch the program, instruments the program's branch transitions, and then encodes and stores the code coverage data in a bitmap. To speed up the process of repeatedly executing the target program, which is necessary for fuzzing, FIRM-AFL uses a mechanism called *fork*. This mechanism runs the target program up to a certain point so that the program's code and data have been properly initialized, and then repeatedly forks a child process from it. Because of this, the parent process is called *fork server*. The input is fed into the forked child instance, and then coverage information is collected and stored in a bitmap that is shared between all three instances (afl-fuzz, fork server, and child instance). Lastly, afl-fuzz uses the bitmap from the child instance to compare it with the bitmap obtained from all past executions to determine if this mutated input should be kept as a new seed and stored in the seed queue [43]. The GitHub repository of Firm-AFL has not been updated since October 2020 [42]. In the repository there is information on how to install and use the fuzzer with the commands to run it. It does not mention a recommended environment, configuration, or output analysis.

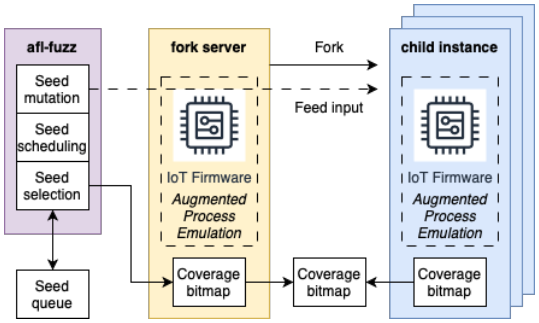

Fig. 4 Overview of FIRM-AFL, adapted from [43].

FIRMCORN [16] is a grey-box fuzzer uses a vulnerable-code search algorithm to determine the entry point for vulnerability-oriented fuzzing and optimized virtual execution technology to improve execution speed, accelerate execution accuracy, and ensure execution stability. The emulator is based on Unicorn Engine [5] and their own fuzzer and crash submodules. An overview of the architecture is modelled in 5. The framework is separated into five parts: preanalysis, dump, fuzz, hook and crash. These five submodules are all connected to FIRMCORN core. The first part is the preanalysis, where an algorithm groups all functions in the firmware based on complexity. The second part is dump, where shell access to the device is obtained in one out of three ways: Telnet/SSH, Device Debug Interface or using the Firmware Update Mechanism. Once shell is obtained, gdbserver is uploaded to the device and this is used to dump context. The hook part is where the Global Offset Table Hook (GOT) information is analyzed. This includes unresolved functions, hardware-specific functions, custom functions and unnecessary functions. The optimized virtual execution part takes place in the core. Here, we can find the CPU emulator, fuzzing policy, the initial environment and a module for heuristic optimization that communicates with the hook part. The fuzz submodule contains the fuzzer, and the crash submodule checks and stores information about crashes [16]. The GitHub repository has not been updated since April 2022 [15]. The repository contains only a link to the technical paper of FIRMCORN. In the paper, there is no information on how to install the fuzzer, run the fuzzer, configure the configuration file, or analyse the output.

EM-fuzz [14] is a grey-box fuzzer that integrates real-time memory checking into fuzzing to improve bug discovery of the fuzzing [14]. It is based on QEMU [29] for emulation and AFL [41] for the fuzzing engine. EM-fuzz aims to reach certain program branches early and execute specific blocks of code more frequently by mutating the input data [14]. It also checks for memory-related vulnerabilities by monitoring the system's memory usage during the execution of the test cases. An overview of the architecture is modelled in Fig. 6. EM-fuzz contains two main modules. The first module is the memory

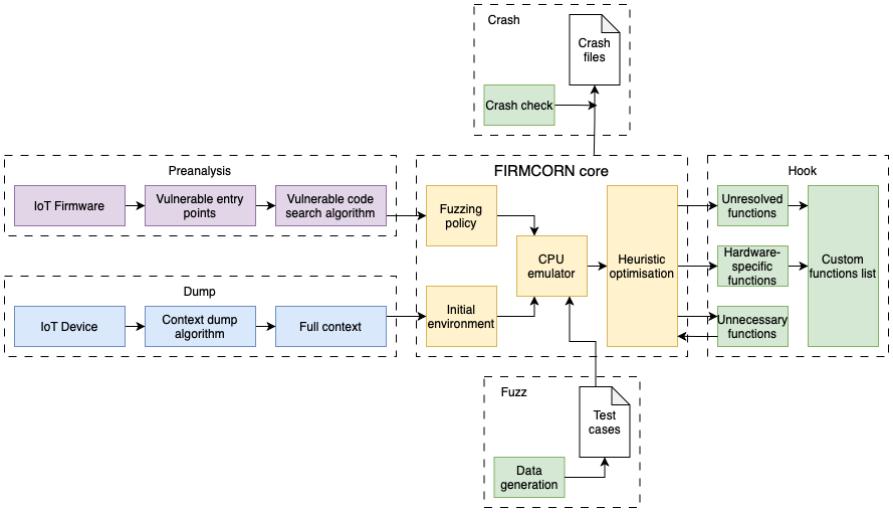

Fig. 5 Overview of FIRMCORN, adapted from [16]

checking instrumentation component which is responsible for recording memory operations that are used in the second main module that performs the guided fuzzing. It uses information about the branches to generate seeds that aid in discovering memory regions that otherwise could be difficult to reach. The parts of the fuzzing module are the fuzzing controller, the seed collector, mutation engine and a process execution component. The controller uses the feedback from earlier executions to have control over which branches have been traversed. The seed selector is responsible for calculating probabilities and adjusting prioritization for the seeds resulting in an execution of a rare branch, ensuring that also rare branches have a high probability of being traversed. The mutator is responsible for maximizing the branch coverage. The process execution component performs the fuzzing, and forwards information on interesting test cases and crashes to their respective queues, and delivers feedback to the controller. The last component is the emulator, which is based on QEMU [14]. As of January 2024, this framework is not open source [40].

FirmHybridFuzzer [38] is a hybrid fuzzing tool that enables testing of MCU firmware without relying on specific peripheral hardware [38]. FirmHybridFuzzer can efficiently test the correctness of a certain portion of the code and maximize code coverage [27]. It is a hybrid fuzzer, which means that it combines the benefit of both traditional fuzzing and symbolic execution [38]. The key solution in this fuzzer is the implementation of a virtual peripheral, allowing it to emulate the behaviour of different peripherals. FirmHybridFuzzer uses AFL [41] as the fuzzer. Fig. 7 shows an overview of FirmHybridFuzzer. The input to the fuzzing framework consists of a firmware binary and the output is the potential firmware vulnerabilities. FirmHybridFuzzer works on the firmware as a whole, meaning it tests both the application tasks and

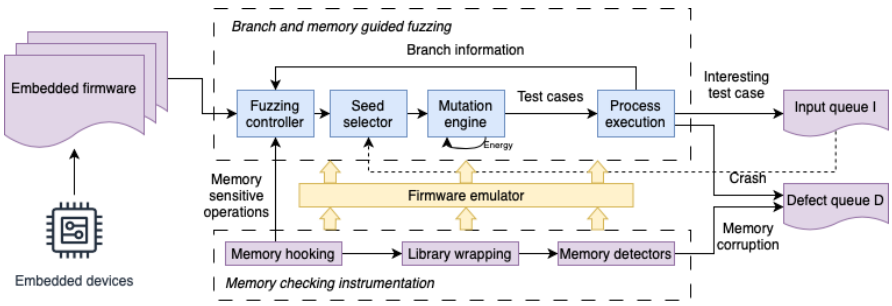

Fig. 6 Overview of EM-fuzz, adapted from [14].

the kernel. To do so, the fuzzer uses the symbolic peripheral-based execution technique. That is a technique to be able to execute firmware in the lack of a physical peripheral and hybrid event generation. The fuzzer also uses a mechanism called multiple dimensional coverage feedback based guidance to improve test case generation. To identify common firmware vulnerabilities, FirmHybridFuzzer implements a fault detection mechanism [38]. The last update on the GitHub repository was November 2023 [37]. The documentation contains information on how to install and run the fuzzer. They do not provide all the installation commands, but refer to README files from other repositories. Only brief information on basic usage and configuration is included. Output analysis is not explained [37].

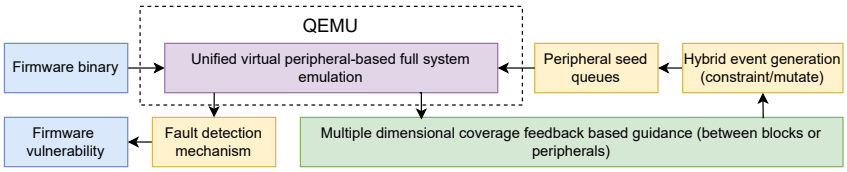

Fig. 7 Overview of FirmHybridFuzzer, adapted from [38].

PRETENDER [19] is a fuzzer that observes the interaction between the hardware and firmware to automatically create models of peripherals. What sets this apart from the other fuzzers, is that they are stateful, interactive and transferable. By creating these models of peripherals, the firmware can be executed in a fully virtual environment. The emulator is based on QEMU. It only supports ARM Cortex devices [19]. An overview of the architecture is modelled in Fig. 8. PRETENDER works on unmodified firmware by observing interactions within the hardware and generating models based on the hardware peripherals. This can be observed by the input to the emulated environment in the figure. This is a hardware-in-the-loop technique where the

models that are generated are fed into an emulator, in this case QEMU. This is then processed using machine learning and pattern-based analysis, and directed back to the emulated environment [19]. The GitHub repository was last updated in August of 2020 [18]. The README file contains information on how to set up, install and run the fuzzer. Dependencies are mentioned. Commands for running the fuzzer is explained, but there is no information on how to configure the targets or analyse the results [18].

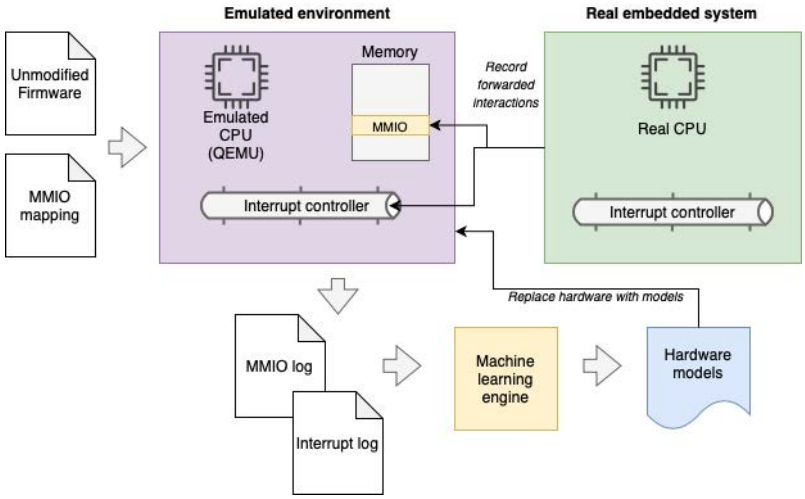

Fig. 8 Overview of PRETENDER, adapted from [19].

P^{2}IM [12] is a fuzzing framework and presents a technique called Processor-Peripheral Interface Modeling, that models I/O accesses for several peripherals. It treats peripheral devices as black boxes. It uses QEMU [29] for emulation and AFL [41] for fuzzing. P^{2}IM uses a technique called explorative execution to automatically infer acceptable state register values during runtime [12]. Handling state register writes is much simpler and the same for all firmware [12]. An overview of the architecture is modelled in Fig. 9. The fuzzer is an unmodified implementation of AFL, and the emulator based on QEMU. The Processor-Peripheral Interface Modeling technique proposed by P^{2}IM is what models the behaviour of the peripherals. These models check the criteria of a property they propose called Processor-Peripheral Interface Equivalence, which is a prerequisite for executing the emulated firmware. The emulated firmware is executed, and feedback is provided back to the fuzzer [12]. The GitHub repository for this fuzzer was last updated in November 2023 [11]. The documentation includes information on how to install the fuzzer, configure the target, perform the fuzzing and analyse the results. Preparing the target, including configuration, is also explained with references to the paper for more information. Its basic usage, including optional customization,

statistics and coverage is mentioned, as well as some commands that help in output analysis. Separate README-files for adding an MCU, building QEMU and prepping the firmware for fuzzing is referenced. Thorough explanations for each step are not included, and there is a limited information on how to analyze the results [11].

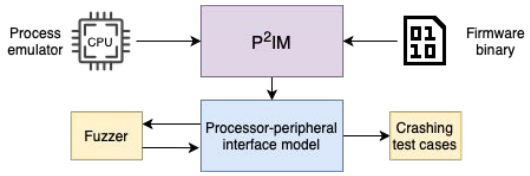

Fig. 9 Overview of P^{2}IM, adapted from [12].

Fuzzware [31] is a fuzzing tool that self-configures inputs to test a firmware image without knowledge about the underlying architecture [31]. It uses access models to transform fuzzing inputs to meaningful values for the firmware to interpret. It uses AFL [41] as the fuzzing engine and supplements it by providing an emulator and access models to translate firmware images to something that is fuzzable by AFL [41] [31]. The emulator is based on Unicorn Engine. An overview of the architecture is illustrated in Fig. 10. The fuzzer generates input that is provided to the emulator. MMIO accesses triggers parts of the raw input to be consumed by the MMIO access models. Here, they are transformed into hardware-generated values and sent to the emulated target. Then, coverage feedback is forwarded to the fuzzer for further mutation [31]. The GitHub repository was last updated in October of 2023 [33]. The documentation includes information on how to install Fuzzware locally or using Docker, how to configure targets, and how to analyse the results. It also includes thorough explainations for each step, each component and an overall explaination of the fuzzer. It links to separate README-files for each step, where configuration, customization and analysis is explained in-depth [33].

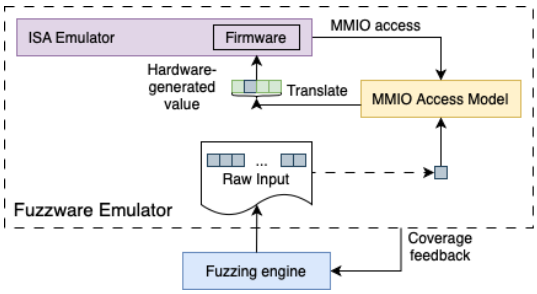

Fig. 10 Overview of Fuzzware, adapted from [31].

Hoedur [32] attempts to mitigate shortcomings of previous black-box fuzzers by allowing the fuzzer to process feedback from firmware execution and understand inputs in a multi-stream form. This is done by deciding the context of the accesses during execution, which involves the counter, the MMIO register address of the request and the size of the access. Based on these values, Hoedur can generate input streams that are combined into a multi-stream input. The fuzzing engine is mainly based upon libFuzzer, with certain features borrowed from AFL and AFL++ [41] [32]. The emulator is based upon QEMU [29], with some minor modifications. Automated DMA support is not yet implemented in Hoedur, but can be manually handled within the MMIO setting. An overview of the architecture is modelled in Fig. 11. The fuzzer provides a multi-stream input to the emulator. The emulator is what exists within the dotted frame in the figure, to execute the firmware using the multi-stream input. During execution of the target firmware, firmware-specific feedback is recorded. The emulator then provides the extended feedback back to the fuzzer [32]. The GitHub repository was last updated in October of 2023 [34]. The documentation includes information on how to install, fuzz, collect coverage information, and run tracing or hooking scripts. It explains for basic usage, and mentions to use the –help for details on customizing the fuzzer. There is no information on how to configure the target or analyse the results [34].

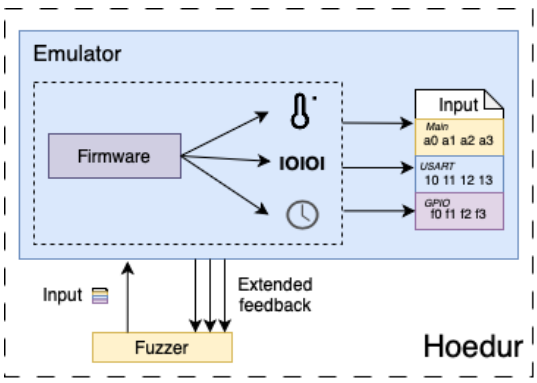

Fig. 11 Overview of Hoedur, adapted from [32].

Ember-IO [6] is a fuzzing framework for firmware that uses model-free memory mapped I/O. This means that the fuzzing framework manages the complexities posed by fuzzing a wide range of memory mapped peripherals without the need to generate register models. It was an initiative to enhance the capabilities of existing fuzzing tools for firmware. Ember-IO uses AFL++ for the fuzzer and QEMU as the emulator to re-host monolithic firmware [6]. This fuzzing framework is focused on monolithic firmware and supports firmware built for ARM Cortex-M microcontrollers. The Ember-IO

paper introduces *functional equivalent coverage instrumentation* (FERMCov) and *peripheral input playback* (PIP) techniques. These techniques improve the performance of monolithic firmware fuzzers by enabling a more efficient search through the input-space and thus overcomes the obstacles imposed by MMIO when accessing and reaching deep code paths [6]. The PIP technique provides mutations more likely to reach previously unseen code and FERMCov increases the likelihood of the mutation efforts on inputs to uncover interesting code paths [6]. An overview of the architecture is modelled in Fig. 12. The MMIO manager component is in control of the raw input stream that is sent to the firmware. The FERMCov generator is the component that sends coverage feedback to the fuzzer, and is responsible for the functional equivalent coverage instrumentation. The peripheral memory store within the MMIO manager is responsible for the PIP technique. The interrupt manager sends an interrupt if a sleep instruction is triggered, and also at certain times based on how many blocks are executed [6]. The GitHub repository of Ember-IO was last updated June 2023 [8]. The documentation includes information on installation, fuzzing, configuration and analysis. Configuration is explained. For the analysis part, replaying inputs and debugging crashes is explained. The GNU Project Debugger (GDB) can be used as well by adding some flag to the command. This option has its own file in the repository, where it is explained in depth. The README file also includes some details on Peripheral Input Playback and FERMCov, which can be useful to understand how the fuzzer works [8].

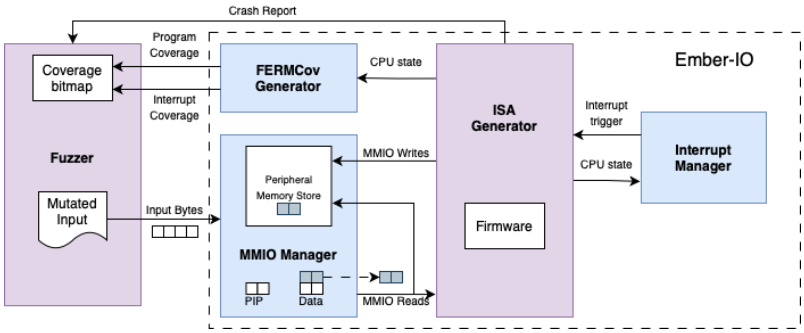

Fig. 12 Overview of Ember-IO, adapted from [6].

μEMU is a firmware fuzzer that is based on S2E, which is a QEMU-based concolic execution tool [45]. μEMU is an approach to emulate firmware with unknown peripherals. It works by taking the firmware image as input and then symbolically executes it by representing unknown peripherals registers as symbols [45]. Fig. 13 shows the architecture of the fuzzer. The fuzzing engine is based on AFL, but is modified to use AFL only for test-case generation and FuzzerHelper for the rest. This includes coverage instrumentation,

the fork server, and crash/hang detection. FuzzerHelper is modelled in the Plugins section of Fig. 13. Since S2E 2.0 does not support ARM MCUs, μEMU has made two contributions to include support for ARM architecture: porting ARM dynamic binary translation to the S2E emulation using QEMU, and adding interfaces for QEMU for managing the CPU of ARM Cortex-M through the kernel-based virtual machine (KVM) interface. The GitHub repository was last updated November 2023 [44]. The documentation contains information on installation, including optional methods and recommended environment, fuzzing, configuration and analysis. It includes a subsection on how to update the repository, which is not included in many of the other READMEs. Usage is explained thoroughly, with steps to configure the target and to run the fuzzer. Analysis is explained to some extent, also how to calculate coverage and use GDB to debug. Configuration is explained well in a separate file [45].

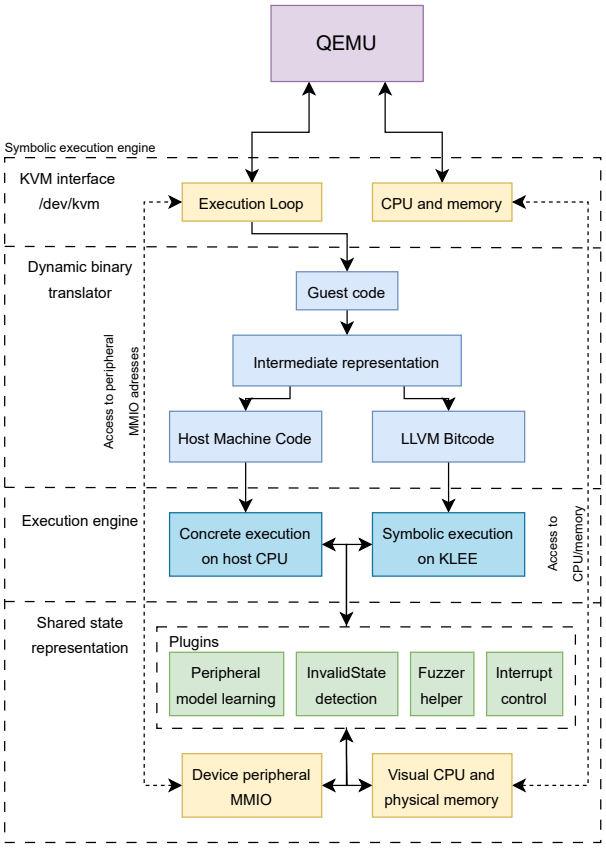

Fig. 13 Overview of μEMU, adapted from [45].

SplITS [7] is implemented on top of Fuzzware, and extends it by running a feedback guided search for input-to-state mappings as well as provide a strategy for optimized replacement of input. The framework also applies the FERMCov technique described in Ember-IO [6] to increase the effectiveness of the framework. Embedded devices use input-to-state mappings to bypass problems like infinite loops that can occur if a peripheral does not respond with an expected response in string format (OK). SplITS allows the mapping and replacement of non-contiguous strings in input-to-state transitions. The overview of SplITS is shown in Fig. 14. SplITS is based on Fuzzware, therefore using Unicorn Engine as the emulator, and extends it by implementing AFL++[41] as the fuzzer. String comparison is performed in the instrumentation and feedback part of the emulator. Length feedback is also performed here, by using the fuzzer's coverage bitmap. The feedback guided search and replacement component receives information about string comparisons and position of the mutated bytes. Then it performs the comparison by using incremental feedback [7]. The GitHub repository is last updated in September 2023 [9]. The documentation includes information on installation, fuzzing and an overview of the architecture of the fuzzer. It mentions a recommended environment, dependencies, and explains basic usage. It does not contain any further information, but since the fuzzer is built on top of Fuzzware, the documentation from Fuzzware is also included in the repository. Like explained earlier in this review, Fuzzware is thoroughly documented, and includes information of configuration, customization and analysis.

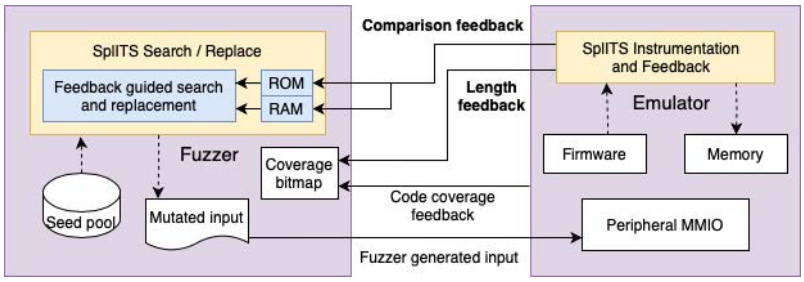

Fig. 14 Overview of SplITS, adapted from [7].

Jetset [22] uses guided symbolic execution to generate a model of the device's peripherals and passes it on to QEMU [29] to emulate the hardware. AFL is used as the fuzzing engine. To run Jetset, developers need access to the executable code, the memory layout, the entry point address and the program goal address. Jetset has no support for devices using DMA because these devices gains access to the memory without consulting the CPU. Therefore the firmware cannot observe these accesses [22]. The architecture of Jetset is modelled in Fig. 15. Jetset uses the firmware binary, its entry point and memory map as input. Directed symbolic execution is performed in the de-

vice interference stage to discover interesting locations in the firmware for further analysis. In the next step, the device synthesis, constraints discovered in the previous step is used to construct a model of the device. Lastly, this model is used in QEMU to emulate the firmware. AFL is used for fuzzing [22]. The GitHub repository of this fuzzer was last updated March 2022 [21]. The documentation contains information on installation, fuzzing, reproducing the crashes as well as links to related repositories and FAQs. It mentions dependencies and shows the basic usage with explanations. It also includes information on how to reproduce the results, but does not contain any additional documentation on how to analyze and interpret the results. It does not include any background information or information on how to configure or customize. It also does not recommend a specific environment [21].

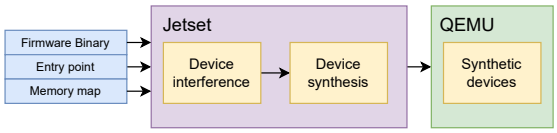

Fig. 15 Overview of Jetset, adapted from [22].

FIRMNANO [17] is a fuzzing framework based on augmented virtual execution, similarly to Firm-AFL (3), to implement fuzzing for firmware. FIRMNANO uses MMIO modeling, implements a virtual interrupt controller, and supports emulation of DMA peripherals and controllers [17]. The fuzzer is AFL [41] and afl-unicorn, which allows fuzzing of binaries emulated with Unicorn Engine [39]. The architecture of FIRMNANO is shown in Fig. 16. The emulation takes place in the augmented virtual execution core. Within the core is the CPU emulator and a component for MMIO modelling, interrupt handles and DMA support. The emulator is based on Unicorn Engine. The MMIO model part records and adds information about peripherals to a *peripheral_list*. During the pre-exection, this list is traversed to map the regions for MMIO. During execution, these are provided to the Unicorn Engine API. The interrupt handle part involves a virtual interrupt controller that streamlines the emulation of the interrupts. The DMA support part uses information about firmware symbols to emulate DMA-related functions. After the firmware has been emulated in the core, it is fuzzed by AFL. The outputs are mutated and directed back towards the core for execution on the emulated firmware [17]. As of March 2024, we haven't received a response from the paper's authors regarding the availability of the tool as open source, despite our attempts to contact them.

HD-Fuzz [23] is a firmware fuzzer developed to be aware of the underlying hardware dependencies by utilizing a hybrid MMIO modeling scheme.

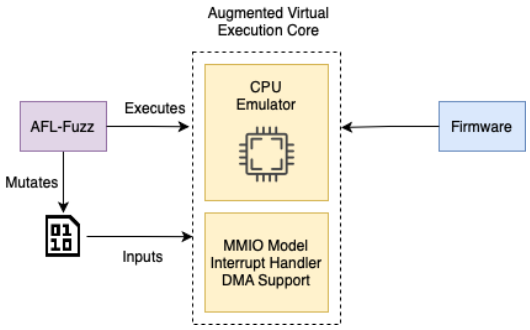

Fig. 16 Overview of FIRMNANO, adapted from [17].

This is a technique where fuzzing and symbolic execution is adapted to the firmware. It is implemented on top of a QEMU-based [29] platform for symbolic execution, S2E 2.0 [2], and uses AFL [41] for fuzzing. HD-Fuzz is unable to observe DMA accesses, and therefore cannot emulate these responses [23]. Another limitation of HD-Fuzz is per-firmware manual effort. As depicted in Fig. 17, the workflow of HD-Fuzz is divided into 4 parts: inputs, rehosting, hybrid MMIO modeling and outputs. The first step is to configure and input the firmware to the rehosted system. Here, the firmware is separated into *system-side firmware code* and *user-side firmware code* to execute them sequentially. To include MMIO modeling, a hybrid solution is implemented. The system-side code is used for initial modelling, then the user-side code is used for mutational modeling using the hybrid method. The additional search and is used to deduce MMIO responses. HD-Fuzz uses these steps to create inputs: crashing inputs and peripheral inputs. Lastly, these are passed to the fuzzer and the results re-iterated through the hybrid MMIO method to discover new paths [23]. As of March 2024, we haven't received a response from the paper's authors regarding the availability of the tool as open source, despite our attempts to contact them.

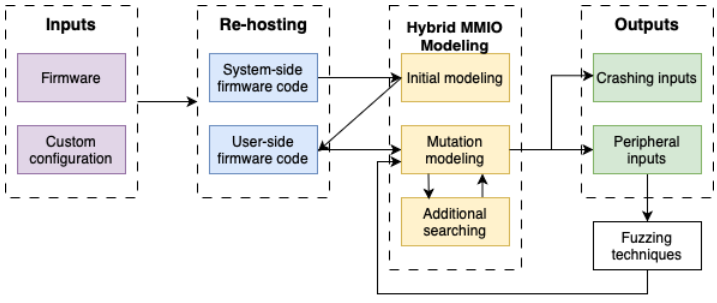

Fig. 17 Overview of HD-Fuzz, adapted from [23].

SAFIREFUZZ [36] implements a different method of rehosting than many of the competitors. It is a technique they call *near-native rehosting*. They have based the peripheral interaction emulation on the approach used in HALucinator (3), using HAL-based hooks [36]. Instead of rehosting a virtual image of the firmware, they adapt binaries to be rehosted in an ARM Cortex-A system that is more powerful [36]. It uses LibAFL as the fuzzing engine. The engine can only run on ARMv7-A cores, e.g. to be found in Cortex-A cores in Raspberry Pis. SAFIREFUZZ does not rely on an emulator and is thus not emulating a whole architecture [36]. This is to avoid the performance penalty induced by emulators such as QEMU. Instead, it uses a light-weight dynamic binary rewriting approach called near-native rehosting. This technique exploits that some ARMv8-A cores provide userspace compatibility with the AArch32 and Thumb instruction set variants [36]. This is done instead of emulating the firmware through lifting and recompilation. Like modelled in Fig. 18, SAFIREFUZZ implements a fuzzing engine, LibAFL, that executes the target, is responsible for rehosting and instrumentation. High-Level Emulation using HALs is used to handle unknown peripherals, by implementing a harness that is tailored to the firmware. This is responsible for memory initialization and records HAL-hooks, and does this by modifying the initial basic block. Translation of these blocks is performed by the engine, while at the same time instrumenting the target, moving the execution of the correct hooks and estimate interrupts. To ensure efficiency, only one basic block is rewritten at a time. This is to avoid multiple rewrites to blocks, and reduce overhead in instrumentation [36]. The GitHub repository was last updated in September of 2023 [35]. The documentation includes information about installation, usage, harnessing and performance. It specifies recommended environment, as well as packages and libraries. Commands to run the fuzzer are provided and briefly explained. There is no documentation on analysis and interpretation of the results. Overall, the documentation for this fuzzer is short and lacks explanations and options for customization. It mentions where to specify the configuration file, but does not contain any information on how to properly configure the target [35].

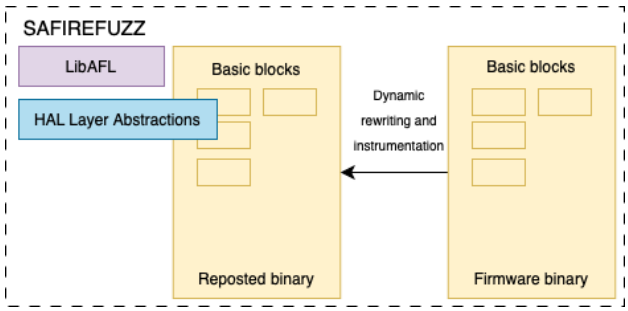

Fig. 18 Architecture of SAFIREFUZZ, adapted from [36].

4 Discussion

After having examined the candidate tools, we will now discuss the differences and similarities between them. Apart from the technical differences of the fuzzers, it is important to consider ease-of-use to evaluate the different fuzzers' applicability in real-word firmware security testing. Therefore, we will begin by discussing the non-technical details of the fuzzers, and then proceed with the architectural and functional differences.

The most important aspect of the ease-of-use of a fuzzer is the documentation, both for installation and further usage. Most fuzzers in this review included information on how to install the tool, but there are few fully developed implementations for firmware developers to use out-of-the-box. If there isn't enough guidance on how to use fuzzing in different situations or on how to understand its results, it will be more difficult to adopt fuzzing as a new method for security testing.

The only fuzzers in this review that have thorough and descriptive documentation for installation, usage and analysis, are Fuzzware, Ember-IO and μEMU. SplITS indirectly includes documentation as it uses Fuzzware as a submodule, and therefore uses the same commands. Because of this, Fuzzware's documentation can be used for fuzzing with SplITS.

Another important aspect when it comes to comparing the fuzzers is whether it is actively maintained and up-to-date with architectures and technologies. Firmware fuzzing is a relatively new concept, and a large portion of the research that exists on the topic has been conducted in the last couple of years. Therefore, we expect new methods or improvements to existing methods to be discovered rapidly. Choosing a tool that is actively maintained could ensure that the latest technologies or methods are being used. It also makes it more likely that any issues will be replied to and possibly fixed.

A third important criterion to ensure ease-of-use in commercial devices, is for the licensing to allow the tool to be used in a commercial environment. Developers need be allowed to run the code and use the tool. While working on this review, we contacted the developers of fuzzers that were missing appropriate licensing, and licences were added to their repositories. We did not receive a response from FIRMCORN, which is still missing a licence.

AFL has grown to become the industry standard for fuzzing software. Amongst the reasons why, are low-effort testing due to automation, simplicity of design, and compatibility with emulators to expand the support to more than just software. The original AFL fuzzer has been bypassed by a new version, AFL++, which provides additional features and options for customisation [13]. Out of the fuzzers we have included in this review, only SplITS and Ember-IO implements AFL++ directly. Some of the other fuzzers have the option to use AFL++, however it must be set specifically. Another difference is whether the fuzzers use an out-of-the-box implementation of AFL, or use it with modifications. Some of the fuzzers (µEMU, P2IM and Fuzzware) use AFL unmodified, as a component solution in their products. Most of the

remaining fuzzers modify the fuzzing engine to fit their use cases [32]. As discussed above, Hoedur implement their own fuzzing engine based on libFuzzer [32].

When it comes to emulating the firmware, almost all the fuzzers in this review use QEMU directly or a QEMU-based implementation. Fuzzware, FIRMCORN, SplITS, and FIRMNANO use Unicorn as the emulator, and Unicorn is based on QEMU. A fuzzer that stands out from the rest is SAFIREFUZZ, as it does not use an emulator, but the near-native rehosting technique.

As of 2024, complete emulation of the varieties of MCUs is not yet possible. There are two techniques for testing firmware: hardware-in-the-loop and emulation for virtually rehosting the hardware. Out of all the candidate fuzzers, only μAFL uses hardware-in-the-loop. By using this method, it bypasses the need for abstractions by forwarding hardware accesses to a physical device during emulation [25]. Although this makes it possible to analyse firmware dynamically, this method is not very useful for fuzz testing. It requires one instance of the target hardware per fuzzing thread since the hardware state and fuzzer must be maintained consistent, in addition to the fact that forwarding is typically a bottleneck for most of these systems [31].

The remaining fuzzers discussed in this review apply different techniques to emulate the hardware. Some of them replace the interactions in the Hardware Abstraction Layer, including HALucinator and SAFIREFUZZ. µEmu and P^{2}IM use heuristic models for automating the same process. A third technique is to perform the fuzzing on all the registers and use other methods to reduce overhead. This is done in Fuzzware and Ember-IO. Lastly, techniques such as symbolic execution and coverage-guidance is used in several of the fuzzers to make the process more efficient [7]. An alternative to symbolic execution is developed in Ember-IO to create a fuzzer that does not generate models as part of the emulation. This is done to avoid the increase in manual labor to avoid failures caused by misrepresentations in heuristic-based models. It can also include testing of error handlers, which is known to cause a significant amount of bugs. This is the FERMCov technique described in the review. This technique is applied in some of the other fuzzers as well: SplITS and experimentally on Fuzzware [7].

HALucinator and SAFIREFUZZ abstracts away the hardware interface using a technique that enables the use of fuzzers like AFL [41] without having to perform any peripheral functions. While these HAL-based fuzzers are unable to operate on the driver level, μAFL targets these specifically. This is different from the other approaches, as drivers are peripherals that interact with the outside world and should be tested accordingly [25].

While reviewing the papers, we discovered that several of the authors are involved in more than one of the fuzzers in this review. This could be a reason as to why many of the fuzzers have similar techniques, components or even build directly on top of each other. In the following we will discuss some similarities between the fuzzers.

Fuzzware and Hoedur are created by almost the same research team, where Hoedur is a strictly improved version of Fuzzware. Both are built using elements from AFL/AFL++ as their fuzzing engine. One major difference in their architecture is the emulator. Fuzzware uses Unicorn and Hoedur uses QEMU. The reason for Hoedur not using the same emulator as Fuzzware is that QEMU makes it easier to support hardware features [30]. However, they are also similar because Hoedur includes the Fuzzware modeling, in addition to use the multi-stream inputs.

Ember-IO and SplITS are created by the some of the same developers, and was intended to target two different types of firmware. Ember-IO was developed to cover the repetitive access pattern that is the case for many peripherals. Therefore it is beneficial for fuzzing firmware that performs a large amount of input and output operations. SplITS is intended for firmware that performs string comparisons, and applies Input To State (ITS) mappings to do so. Like mentioned in the review, Ember-IO is built using AFL++ and QEMU while SplITS is built on top of Fuzzware which uses AFL and Unicorn. This is because the Peripheral Input Playback (PIP) technique applied in Ember-IO led to a slight decrease of consistency in performance - some data could be repeated by the PIP technique, meaning that a byte in the firmware does not necessarily correspond to a single byte in the fuzzer generated input [10].

Another difference is that SplITS depends on a one-to-one mapping between bytes from the observed firmware buffer and the fuzzer generated input bytes for input-to-state mapping and replacement. If PIP is applied to the data values that are loaded into the firmware buffer, this can create a one-to-many mapping, preventing replacement with a single suitable value. While SplITS' techniques can be applied to Ember-IO, the developers of SplITS found the aforementioned issue would occur in some fuzzer generated inputs, preventing SplITS from being applied successfully to those inputs, reducing SplITS' overall consistency on Ember-IO compared to Fuzzware [10].

Pretender [19] and P^2IM both use automation of the emulation procedure by using direct MMIO modeling. They do this in two different ways: Pretender uses recordings of real activity in the MMIO region of the device to model the peripherals, while P^2IM uses *blind fuzzing*, which is corresponding to black-box fuzzing. Both methods have advantages and disadvantages, the method used in Pretender requires access to the physical device while the method used in P^2IM is less generic and will not produce an accurate model in many devices [4].

One of the most difficult aspects of analyzing the fuzzed firmware is the false positive rate. A false positive is a "false alarm" and occurs when the fuzzer incorrectly identifies parts of the firmware as vulnerable. A high false positive rate diminishes the effectiveness of the fuzzing tool, as it means spending more time sorting through false alarms rather than focusing on actual vulnerabilities, which can be frustrating and time-consuming. Thus, triggering crashes does not need to mean that it represents a security issue.

An aspect to consider when discussing false positives found while fuzzing a firmware image of an embedded device, is the hardware constraints. As discussed in the background section, developers have to consider if vulnerabilities are likely to be exploitable if the code is not reachable on the physical device. Even though the optimal solution is to have the firmware secure in all environments, this would be a question of prioritization for the development team. Ember-IO mentions that two out of the 6 new bugs discovered by the fuzzer could be vulnerabilities that would not be reachable on the physical device. Even though this does not count as a false positive, the development team of the device might not expect to trigger them, unless the embedded device is defective.

The issue of false positives arises partly because inputs for a particular register is defined by unknown hardware [6]. This makes it challenging to deduce directly from the firmware. For fuzzers like P^{2}IM and μEmu, this is unlikely to happen because in these approaches they enforce register models based on common access patterns. In these fuzzers some registers can only contain values written by firmware and thus prevents unexpected inputs. Other fuzzers, such as Fuzzware and Ember-IO, have a different approach where all registers are considered as valid sources to insert the fuzzing input.

Some false positives can be linked to the lack of DMA support in several of the fuzzers. DMA enables the RAM to be accessed by the peripherals directly without running it through the processor. Therefore, using symbolic execution is not useful in discovering information about the DMA accesses [45]. We will discuss this further below.

DMA is mentioned and explained in several of the papers, some including options on how to incorporate support for DMA input generation. Amongst these are manual configuration of MMIO settings and installing separate components on top of the fuzzer. As explained, not including DMA support can cause an increase in the amount of false positives [23], but none of the fuzzers in our study include automated DMA support at the moment.

The research team behind Ember-IO and SplITS have followed up with a supplementary component, DICE, to better accommodate fuzzing of firmware that utilizes DMA [12]. The authors mention the option to implement DICE on top of existing fuzzers to include automated DMA support [6]. DICE [26] is a component to extend existing firmware analyzers which enables them to create or modify DMA input channels (from peripherals to firmware) [26]. As soon as the firmware inserts the source and destination DMA transfer pointers into the DMA controller, DICE recognises DMA input channels [26]. Subsequently, DICE modifies the data transmitted via DMA on behalf of the firmware analyzer.

The only fuzzer included in this review that already includes support for DMA is HALucinator. This is due to the HLE techniques applied in the emulation of the firmware. HALucinator handles DMA by using the same HALs as the developers use to perform DMA [4]. Here, the DMA accesses are removed from the program. P^{2}IM however, only considers MMIO interaction

sequences as input, so if a crash is found it must be mapped back to the corresponding input. In addition, the inputs are replayable against virtualized and actual targets [4].

For the fuzzers without DMA support, a possible way of incorporating support for DMA is by configuring a simple DMA overlay in the memory maps [30]. This is done by manually setting an MMIO overlay region to allow manually specifying DMA buffers in case they are located in RAM. The resulting behavior is that fuzzing input is provided each time the specified region is accessed.

Table 1 Overview of the firmware fuzzing tools mentioned in this review.

	FUZZER	**EMULATOR**	**TECHNIQUE**
HALucinator [4]	AFL	QEMU	HAL-based
μAFL [25]	AFL	QEMU	Hardware-in-the-loop
FIRM-AFL [1]	AFL	FIRMADYNE	Coverage-feedback [40]
FIRMCORN [16]	Their own implementation	Unicorn	Pattern-based
EM-Fuzz [14]	AFL	QEMU	Coverage-feedback [40]
FirmHybridFuzzer [38]	AFL	QEMU	Symbolic Execution
PRETENDER [19]	Their own implementation	QEMU	Pattern-based
P^{2}IM [12]	AFL	QEMU	Pattern-based
Fuzzware [31]	AFL	Unicorn	Symbolic Execution
Hoedur [32]	libFuzzer	QEMU	Symbolic Execution with Multi-stream input
Ember-IO [6]	AFL++	QEMU	Coverage-feedback (FERM-Cov)
μEMU [45]	AFL	QEMU (S2E)	Symbolic Execution
SplITS [7]	AFL++	Unicorn	Symbolic Execution with FERMCov [6]
FIRMNANO [17]	AFL	Unicorn	Coverage-feedback
HD-Fuzz [23]	AFL	QEMU (S2E)	Symbolic Execution
Jetset [23]	AFL	QEMU (S2E)	Symbolic Execution
SAFIREFUZZ [36]	LibAFL	Near-native rehosting[2]	HAL-based

5 Conclusion

Even though the research field on fuzzing embedded devices is growing, firmware fuzzing is still so complicated that there is a long way to go until we can expect adaptation in most security testing routines of embedded devices. An important step in the right direction is ensuring thorough and

[2] Safirefuzz does not rely on an emulator.

descriptive documentation, which is why we have decided to emphasise documentation in this review. If this ensures adaptation in real-world projects, this could lead to more feedback that the developers of the fuzzers can use to improve the tools. Several of the papers in this review discuss the possibility of future support for DMA. After communicating with the developers of the different fuzzers, our impression is that DMA support is a priority in future work for most of them.

Table 1 shows a summary of all the fuzzers, visualizing the similarities and differences between them. The table shows which fuzzing engine, emulator and technique each of the firmware fuzzers use.

Acknowledgements

The research in this paper was co-funded by the European Union through the Horizon Europe NEMECYS project, grant number 101119747.

References

1. Chen, D.D., Woo, M., Brumley, D., Egele, M.: Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS (2016)
2. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. ACM Sigplan Notices **46**(3), 265–278 (2011)
3. Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M.: Halucinator. URL `https://github.com/sandialabs/halucinator`
4. 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 Security 20), pp. 1201–1218. USENIX Association (2020). URL `https://www.usenix.org/conference/usenixsecurity20/presentation/clements`
5. Engine, U.: The ultimate CPU emulator. URL `https://www.unicorn-engine.org`
6. Farrelly, G., Chesser, M., Ranasinghe, D.C.: Ember-IO: Effective firmware fuzzing with model-free memory mapped IO. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS '23, p. 401–414. Association for Computing Machinery, New York, NY, USA (2023). DOI 10.1145/3579856.3582840. URL `https://doi.org/10.1145/3579856.3582840`
7. Farrelly, G., Quirk, P., Kanhere, S., Camtepe, S., Ranasinghe, D.: SplITS: Split input-to-state mapping for effective firmware fuzzing. In: European Symposium on Research in Computer Security, pp. 290–310 (2023). URL `https://www.researchgate.net/publication/373142220_SplITS_Split_Input-to-State_Mapping_for_Effective_Firmware_Fuzzing`
8. Farrelly, G., Quirk, P., Kanhere, S.S., Camtepe, S., Ranasinghe, D.C.: Ember-IO. URL `https://github.com/Ember-IO/Ember-IO-Fuzzing/tree/main`
9. Farrelly, G., Quirk, P., Kanhere, S.S., Camtepe, S., Ranasinghe, D.C.: SplITS. URL `https://github.com/SplITS-Fuzzer/SplITS`
10. Farrelly, G.D.: personal communication

11. Feng, B., Mera, A.: P2IM. URL https://github.com/RiS3-Lab/p2im
12. Feng, B., Mera, A., Lu, L.: P^{2}IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 1237–1254 (2020)
13. Fioraldi, A., Mantovani, A., Maier, D., Balzarotti, D.: Dissecting american fuzzy lop: A FuzzBench evaluation. ACM Trans. Softw. Eng. Methodol. **32**(2) (2023). DOI 10.1145/3580596. URL https://doi.org/10.1145/3580596
14. Gao, J., Xu, Y., Jiang, Y., Liu, Z., Chang, W., Jiao, X., Sun, J.: EM-Fuzz: Augmented firmware fuzzing via memory checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(11), 3420–3432 (2020). DOI 10.1109/TCAD.2020.3013046
15. Gui, Z., Shu, H., Kang, F., Xiong, X.: FIRMCORN V2. URL https://github.com/S4CH/FIRMCORN-V2
16. 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). DOI 10.1109/ACCESS.2020.2973043
17. Gui, Z., Shu, H., Yang, J.: FIRMNANO: Toward IoT firmware fuzzing through augmented virtual execution. In: 2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS), pp. 290–294 (2020). DOI 10.1109/ICSESS49938.2020.9237719
18. Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., Vigna, G.: Pretender: Automatically emulating hardware. URL https://github.com/ucsb-seclab/pretender
19. Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., Vigna, G.: 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. USENIX Association, Chaoyang District, Beijing (2019). URL https://www.usenix.org/conference/raid2019/presentation/gustafson
20. Jaatun, M.G., Sæle, H.: A checklist for supply chain security for critical infrastructure operators. In: The International Conference on Cybersecurity, Situational Awareness and Social Media, pp. 235–249. Springer (2023)
21. Johnson, E., Bland, M., Zhu, Y., Mason, J., Checkoway, S., Savage, S., Levchenko, K.: Jetset: Targeted firmware rehosting for embedded systems. URL https://github.com/aerosec/jetset
22. 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), pp. 321–338. USENIX Association (2021). URL https://www.usenix.org/conference/usenixsecurity21/presentation/johnson
23. Kim, J., Yu, J., Lee, Y., Kim, D.D., Yun, J.: Hd-fuzz: Hardware dependency-aware firmware fuzzing via hybrid mmio modeling. Journal of Network and Computer Applications (2024). Available at SSRN 4493040
24. Li, W., Shi, J., Li, F., Lin, J., Wang, W., Guan, L.: microAFL. URL https://github.com/MCUSec/microAFL
25. Li, W., Shi, J., Li, F., Lin, J., Wang, W., Guan, L.: μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, p. 1–12. Association for Computing Machinery, New York, NY, USA (2022). DOI 10.1145/3510003.3510208. URL https://doi.org/10.1145/3510003.3510208
26. Mera, A., Feng, B., Lu, L., Kirda, E.: Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy, S&P/Oakland'21 (2021)
27. Pak, B.S.: Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. School of Computer Science Carnegie Mellon University (2012)

28. Pare G, K.S.: Handbook of eHealth Evaluation: An Evidence-based Approach [Internet]. Victoria (BC): University of Victoria (2017). URL https://www.ncbi.nlm.nih.gov/books/NBK481583/
29. QEMU: QEMU wiki. URL https://wiki.qemu.org/Main_Page
30. Scharnowski, T.: personal communication
31. 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. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 1239–1256. USENIX Association, Boston, MA (2022). URL https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski
32. Scharnowski, T., Wörner, S., Buchmann, F., Bars, N., Schloegel, M., Holz, T.: Hoedur: Embedded firmware fuzzing using multi-stream inputs. In: USENIX Security Symposium (USENIX Sec) (2023)
33. Scharnowski, T., et al.: Fuzzware repository. URL https://github.com/fuzzware-fuzzer/fuzzware
34. Scharnowski, T., et al.: Hoedur repository. URL https://github.com/fuzzware-fuzzer/hoedur
35. Seidel, L., Maier, D., Muench, M.: Safirefuzz. URL https://github.com/pr0me/SAFIREFUZZ
36. Seidel, L., Maier, D., Muench, M.: Forming faster firmware fuzzers. In: Proceedings of the 32nd USENIX Conference on Security Symposium, pp. 2903–2920 (2023)
37. Situ, L., Zhang, C., Guan, L., Zuo, Z., Wang, L., Li, X., Liu, P., Shi, J.: FirmHybridFuzz. URL https://github.com/stuartly/FirmHybridFuzz
38. Situ, L., Zhang, C., Guan, L., Zuo, Z., Wang, L., Li, X., Liu, P., Shi, J.: Physical devices-agnostic hybrid fuzzing of IoT firmware. IEEE Internet of Things Journal **10**(23), 20718–20734 (2023). DOI 10.1109/JIOT.2023.3303780
39. Voss, N.: afl-unicorn. URL https://github.com/Battelle/afl-unicorn
40. Yun, J., Rustamov, F., Kim, J., Shin, Y.: Fuzzing of embedded systems: A survey. ACM Computing Surveys **55**(7), 1–33 (2022)
41. Zalewski, M.: American fuzzy lop (afl) (2023). URL http://lcamtuf.coredump.cx/afl/. Accessed on September 4, 2023
42. Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L.: FIRM-AFL. URL https://github.com/zyw-200/FirmAFL
43. 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. USENIX Association, Santa Clara, CA (2019). URL https://www.usenix.org/conference/usenixsecurity19/presentation/zheng
44. Zhou, W., Guan, L., Liu, P., Zhang, Y.: μEmu. URL https://github.com/MCUSec/uEmu
45. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic firmware emulation through invalidity-guided knowledge inference. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association (2021). URL https://www.usenix.org/conference/usenixsecurity21/presentation/zhou