Overview of IoT Fuzzing Techniques

Tuan-Dat Tran

University of Duisburg-Essen
tuan-dat.tran@stud.uni-due.de

***Abstract*—Due to the rising popularity of IoT devices and embedded systems and their usage in not only in the business sector but also at home, the focus has been shifting to the security of those devices. To address this issue, there have been many approaches in detecting, analysing and mitigating security flaws in IoT devices like static[8] and dynamic analysis[9]. Another approach to vulnerability detection is fuzzing.**

Fuzzing is a technique originally used for automated black box testing software and became a highly researched topic[6][37][35][36], expanding its usage from black box testing to white and grey box testing. Fuzzers generate test cases to test software for vulnerabilities. The generation of those test cases are done in many ways.

IoT fuzzers focus on fuzzing IoT devices. Although there are similarities to regular fuzzing, fuzzing IoT devices comes with its own constraints and techniques.

In this paper, we are comparing techniques used by IoT fuzzers to circumvent the challenges presented by IoT devices and the constraints of the solutions proposed by the IoT fuzzers.

I. Introduction

Internet of Things (IoT) devices and embedded systems are becoming more and more prevalent, and with billions of devices being connected to the internet, they are an integral part of everyday life[16]. Despite IoT devices being so widespread, they are riddled with security vulnerabilities, which makes them an easy target for attackers, since many of those vulnerabilities are considered "low-hanging fruits"[6]. One example of such a vulnerability in IoT devices is the 2016 Mirai botnet which consisted of an average of 200,000 to 300,000 IoT devices[3] while it was suspected that over half a million are vulnerable to the security vulnerabilities the Mirai Botnet utilized[19].

While information leakage[38] and insecure login credentials[3] are just some of the many security problems an IoT device can have, detection and mitigation of these security flaws has proven itself to be challenging. One approach to discover those flaws is called fuzz-testing, or fuzzing.

Fuzzing is a method to test software for flaws by automatically generating and sending large amount of malformed data to the software. This is done while the fuzzer monitors the software's reaction to this data for malfunction like crashes or other unexpected behaviour. The goal of fuzzers is to detect vulnerabilities in the software in an automated manner. Despite the simplistic approach to vulnerability detection, it has proven itself to be effective[21]. The simplistic approach enabled researchers to extend the capabilities of fuzzing from creating test cases consisting of random data to sophisticated systems which use a wide variety of information about the context and source code of the software. The complexity of fuzzing tools goes so far that the techniques used by fuzzers diverge to such an extent that they can be divided into different classes, each with their own strength and weaknesses[22].

A huge advantage of fuzzing compared to other vulnerability detection approaches is the automation of the analysis, which makes fuzzing a highly scalable method to find vulnerabilities[23]. Scaling up the fuzzing process enables a higher rate of generation test cases and therefore increases the coverage on the target program. On the other hand, insufficiently "smart" test generation methods can lead the fuzzer to run for a long time without finding any new code paths or otherwise advancing the fuzzing process[28].

IoT fuzzing applies the methodologies of fuzzing to IoT devices. Just like different kind

of fuzzers, IoT fuzzing has its own advantages and disadvantages. IoT devices offer a large surface area regarding communication, e.g. network protocols, their companion app or their web interface[6][5][35]. For this reason, fuzzers which were not originally designed to fuzz IoT devices can still be utilized for IoT fuzzing, like in the case of boofuzz, which was developed with the intent to fuzz network protocols[5]. IoT Fuzzing also opens the door for new techniques, unique to IoT devices, by fuzzing the companion app of the device[6].

In this paper, we present an overview of different fuzzing tools and techniques for IoT devices. We focus on advantages and disadvantages of those techniques in the context of their use-case to help developers and researchers find the right tool for their job and weigh in the positive and negative aspects of existing approaches. The IoT fuzzing tools, chosen for the overview, were chosen to cover as many recently developed fuzzing techniques to the best of our abilities.

The paper is structured as follows. First we introduce IoT devices, firmware and general fuzzing. In Section III, we lead into IoT fuzzing and its challenges to create a knowledge basis to introduce IoT fuzzing techniques in Section IV, the main section. Section V contains related work, that is closely tied to IoT fuzzing. And finishing up with the conclusion in section VI.

II. Background

A. IoT devices and embedded systems

The terms IoT devices and embedded systems describe a large amount of devices. Embedded systems are devices which interact with their surroundings via sensors and regulators and are built to serve a specific purpose[23]. IoT devices on the other hand are broadly described as devices which extend regular devices with an internet connection to enable them to communicate over the internet[27]. The term embedded devices can describe many devices such as cameras or industrial control systems (ICS), which makes it hard to generalize embedded devices. This also applies to IoT devices, since the extension of an embedded system by an internet connection, makes it an IoT device.

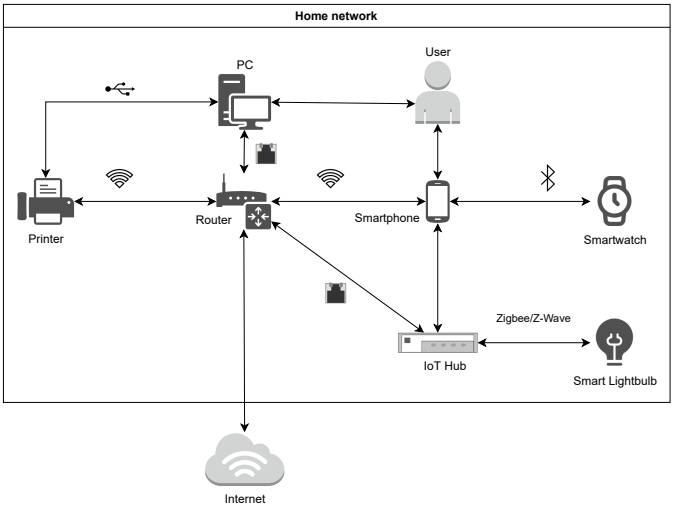

Figure 1. Example of IoT home network (inspred by Wang et al.[35]).

IoT devices, due to being built for specific purposes, do not need as much processing power as a general computer does. This leads to them having a hardware platform specifically tailored to their use case. And due to the heterogenic nature of IoT devices in terms of e.g. operating systems, instruction sets or memory layouts, analysis of the firmware proves difficult[8]. Reasons for this are the different requirements a manufacturer has for the device like the energy efficiency, real-time processing or memory footprint[14].

IoT devices, and especially home-based ones, use multiple ways to connect to the internet. IoT devices connect to the internet either directly through Wi-Fi or via an intermediary device like a smartphone and connecting to it with Bluetooth. Another way is having an IoT hub which acts as proxy between other IoT devices and either another intermediary via Bluetooth or directly Wi-Fi[35]. This leads to many ways an IoT network can be structured, depending on the kind and number of IoT devices (Figure 1).

The works of Hahm et al.[14] propose a classification into low-end and high-end IoT devices and dividing those two classifications into three subcategories for low-end devices. Those classes represent the complexity and computing capability of those devices, with “Class 0” having the least resources and “Class 2” devices having

the most resources. Multi-purpose systems (i.e. smartphones and computers) deploy many mechanisms to detect faults like segmentation faults and report them through core dumps. IoT devices may not have such functionalities. The more minimalistic design of IoT fuzzers causes them to only perform the specific tasks they were built for. Therefore, functionalities like heap hardening may not be present due to the IoT device's limited computing power and constrained costs[23].

B. Firmware

IoT firmware on IoT devices is the software, that acts as an intermediate between higher level software and the hardware of the device. This functionality is provided by the firmwares simplified interface of lower level functionalities, that can be used by higher level software[13] to communicate with the hardware. Since firmware communicates with many parts of the IoT device, it contains a lot of information about it.

There are several types of firmware based on the type of device they are used in. In the works of Muench et al.[23] devices are classified in "Type-0" to "Type-III" systems:

T0 (Type-0) systems represent multi-purpose systems, which don't fall under the classification of embedded systems or IoT devices.

T1 (Type-1) devices are devices, which use a general purpose operating system, like Linux. The operating system on T1 devices is often modified to be more minimalistic and offer a lightweight user environment like busybox.

T2 (Type-2) devices run on completely customized operating systems which are tailored to the device's use case. In order to save space and computational power, typical operating system functions like a Memory Management Unit may be omitted.

T3 (Type-3) devices run on a single control loop. On these devices, the firmware and the software, which runs the device's functionalities, are a single instance. This leads to a so-called "blob firmware"[30], consisting of the application and system code compiled together.

Like all software, firmware is susceptible to bugs and misconfigurations, which can lead to vulnerabilities[8]. For this reason, analysis tools are needed to find such vulnerabilities. There are several methods to analyse firmware for bugs, but they all have to face the challenge of working around the heterogenity of firmware[8].

To analyse firmware, firmware first has to be acquired. This can be done by downloading it from the vendor's website. An alternative method of acquiring firmware is extracting it from the physical device. This is done by either some kind of debugging port or by reading the flash memory directly. Extracting firmware manually poses a challenge in itself, since debugging ports are not always available on the end product[6][31].

Additionally, firmware is often packed or even encrypted, which poses yet another obstacle for firmware analysis. In some cases, proprietary compression algorithms or encryption make firmware analysis infeasible or even impossible.

C. Fuzzing

Fuzzing describes the process of testing a software for faulty or unexpected behaviour by sending it malformed messages as input[12].

There are multiple types of fuzzing techniques based on the amount of known information about the software: White box, back box and grey box fuzzing. White box fuzzing has complete information about the software's source code. Black box fuzzing on the other hand has no such information, while grey box fuzzing lies in between regarding the available information. Black box fuzzing relies purely on the binary of a program or the program in its already executed state[20]. This leads to back box fuzzers generally creating many unnecessary test cases due to the lack of knowledge about the internals of the target[13]. Another problem with back box fuzzers is the detection of errors. Internal system errors, which may lead to misbehaviour at a later time, can not be easily detected by black box fuzzers as they occur. Black box fuzzers therefore often rely on externally visible exceptions. Advantages of black box fuzzing are the narrow and quick tests due to the limited surface area to target, focusing only on the aspects of the software the user interacts with[18]. Additionally, back box fuzzing may be the only way of fuzzing a target when there is no source code available. White box fuzzers on

the other hand have access to the source code of the fuzzing target. Test cases generated by white box fuzzers are based on the analysis of the given source code. Techniques like symbolic execution or dynamic taint analysis are utilized to increase the efficiency of the fuzzer. In comparison to back box fuzzing, white box fuzzing usually has a higher overhead since the additional analysis is performed on the targets source code[21]. Grey box fuzzers take the middle ground between white and black box fuzzers and only use some information about the internals of the target software to improve the fuzzing process. This may be done by injecting instrumentation to the binary at compile time[7] or by performing lightweight static analysis on the source code of the software[21]. The usage of limited knowledge enables grey box fuzzers to have higher throughput than white box fuzzers, while being more accurate than black box fuzzers. Comparing back box fuzzers with grey box fuzzers or even white box fuzzers is therefore not feasible, due to the different starting conditions and use cases[11].

The basic fuzzing process can be divided into three steps: (1) input generation and sending that input to the software, (2) monitoring the software's behaviour in reaction to the given input and (3) adjusting the input according to the software's behaviour (Figure 2).

During the input generation step, the fuzzer generates and prepares messages according to its generation strategy. Choosing which generation strategy is used depends on the given information or constraints of the system that is being fuzzed. The given information about the fuzzing target differentiates fuzzers into the categories black-, white- and grey box fuzzers.

Monitoring the software's behaviour upon receiving a malformed message as input is another step of a typical fuzzing loop. The monitored behaviour depends on the earlier steps, but after every loop the original program's state should be restored to have an equal ground for all test cases. When the fuzzer looks for XSS bugs or SQL injections, the program will not crash, when such a bug is triggered. This has to be taken into consideration while monitoring the software and therefore other methods of detecting those bugs will have to be used than methods, which are used to detect crashes due to memory errors like buffer overflows. Fuzzers who do not try to trigger crashes usually use the application's answer to the input message to determine whether the test case triggered the event, which was tested for[5]. To monitor a software's crash, the fuzzer can provide instrumentation, with which the tested software is compiled[1], if grey box fuzzing is used. A back box approach could be monitoring for specific output of the software after a malformed input has been sent, or monitoring the status of the network connection for networking capable software. A fuzzers goal is to cover as many parts of the software as possible. Evaluating the coverage of the software is only possible for grey box or white box fuzzers, since they can instrument the code. This metric can be used to guide the fuzzing process of generating input, like in the state-of-art mutational fuzzer AFL[1] and its fork afl++[2].

Another property of fuzzers is their adjustments to the input after a fuzzing loop is done. They are categorized into smart and "dumb" fuzzers. Dumb fuzzers are not aware of the input structure and therefore only try random input, substitutions based on heuristics, delete parts of the input or add parts to the input. This can lead to a lot of test cases, which do not lead anywhere. Another disadvantage is that input generated by a dumb fuzzer may easily be dismissed if a specific input structure is expected. Looking at smart fuzzers, which try to generate valid input based on the software's protocol[5], grammar[15] or model[26]. To perform smart fuzzing the input model must be provided to the fuzzer, which may not be as easily accessible on proprietary devices, although there are ways to derive an input model from a large sample of valid and invalid input.

Advantages of fuzzing are the automation and scalability of the process. This enables fuzzing to run many test cases in a short amount of time, which makes throughput of the fuzzer an important metric in evaluating fuzzers[7]. This is achieved by easily being able to run software concurrently on multiple processors. An alternative way is running the software in a virtual environment[23] and executing the virtual environments concurrently.

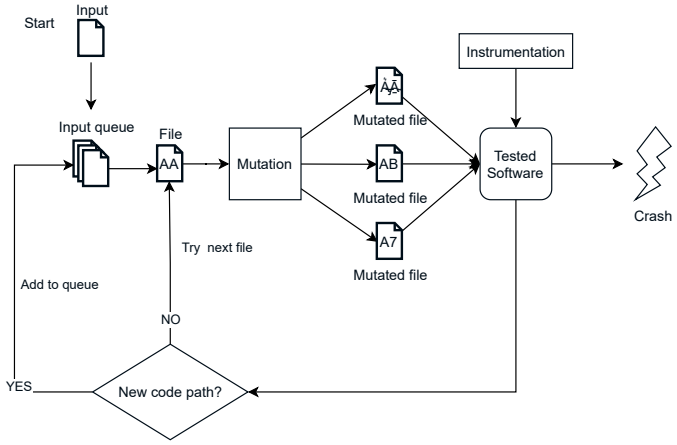

Figure 2. Fuzzing with AFL[1].

III. Challenges of IoT Fuzzing

IoT fuzzing is the application of fuzzing techniques on IoT devices. This approach poses new challenges, since fuzzing hardware and its firmware and fuzzing software operate on different domains, which have each their own challenges.

Muench et al.[23] describe the main challenges: The first challenge, fault detection, is about the complexity of observing crashes on IoT devices during the fuzzing process. Fuzzing regular software may already yield unobservable unexpected behaviour. Working with IoT devices adds another layer to this problem, since IoT devices do not have the same I/O capabilities and memory protection measurements as a multi-purpose system does. The second challenge in IoT fuzzing is the performance and scalability of the fuzzing process. Running a regular fuzzer concurrently on multiple processes rarely poses a challenge. When fuzzing IoT devices, either multiple copies of the same device have to be bought to create a comparable scenario, which is often infeasible or emulation has to be utilized, which poses its own multitude of challenges. The third challenge is instrumentation, used in non-back box fuzzing approaches, to collect code coverage information and detect subtle memory corruptions. There are multiple approaches to add instrumentation to a program or its environment for regular fuzzing. A challenge of adding instrumentation to IoT devices is the need to often use static and dynamic analysis to imitate the functionalities of instrumentation, since they often can not be directly applied to the IoT device. The reasons for this will be explained later.

(1) The challenge of fault detection on IoT devices means, that memory corruptions in the device caused by IoT fuzzers can often go unnoticed since they do not necessarily lead to crashes. Protection measurements on multi-purpose systems detect memory corruptions on the system caused by fuzzing and cause a crash, making memory corruptions therefore visible to regular fuzzers. Such measurements are rarely implemented on IoT devices due to limited computing resources[23].

A liveness check, also called probing, can be performed to check the status of the device while fuzzing it. Probing can either be active or passive. During active probing, the fuzzer sends regular known to be valid messages to the target system and evaluates the response. The messages sent by the fuzzer may cause a state change in the tested device, which has to be accounted for. Passive probing uses the device's responses to the test message to determine liveness or observes visible crashes.

Muench et al.[23] expands on this by classifying system crashes by their observability. An observable crash is therefore the most visible and manageable kind of crash. During observable crashes, the device stops running and provides some form of error message or draws attention to the faulty behaviour in another way. It is added that this also includes crashes, which do not provide additional information about the crash, such as error messages. Observable crashes are the optimal case among system crashes, since they are visible from the outside and enable the fuzzer or tester to react to the crash without delay.

Reboots are another kind of crash. A crash inducing error of a software on an IoT device usually does not lead to the crash of the whole system, since they work indipendently from each other. In T3 devices, where the software and firmware are one and the same, a crash of an outward facing service leads to a crash of the whole system.

In reaction to malformed input, a device may hang. That means that it halts execution and

does not react to any more input. This may be due to being stuck in an infinite loop. This leads to a slowdown in throughput of a fuzzer and the device needs to be restarted if such behaviour is found.

Late Crashes pose a challenge for testing the device. This behaviour is described as the device crashing after a non-negligible amount of time after the real cause of the crash, like a malformed message, is sent. This makes correlation between the cause and the crash challenging.

At last, there are cases where neither the device nor the software crash while still being in an unexpected state, which leads to wrong data or an incorrect output. This kind of malfunctioning of the device is hard to detect, since the fuzzer needs information about the expected response to determine whether it is an output caused by a malfunction or not. This is further complicated due to the diverse message formats used in IoT devices[11].

There are also cases of malformed input not causing any visible effects, even when errors occurred. These errors may cause crashes or malfunctioning at a later time, which makes detecting them during fuzzing almost impossible without instrumentation[11].

(2) The second challenge is performance and scalability. While regular fuzzers can execute and test software concurrently to increase the throughput and therefore find more possible faults in the software over time. Fuzz testing on an IoT device is not possible in the same manner, since a physical device is being fuzzed. Even though multiple copies of the same device could be purchased, to scale up the test cases, it would become infeasible due to financial limitations and infrastructure requirements like power and space. Emulation can help with the problem of scalability by emulating the test device, but this approach faces the challenge of IoT devices being dependent on the hardware components of the device[39].

After a fuzzing loop, the original state of the tested device has to be established to start every fuzzing attempt under the same conditions. This is not a challenge with regular software, since the software's original state is re-established after rerunning it. Changes on the file system, that were caused by the tested software can be easily reverted with e.g. a snapshot of the virtual machine running the test. To establish a testing condition on IoT devices, without the knowledge of its internals, the easiest method is restarting an IoT device. This step can take up to a minute, which negatively affects the throughput of IoT fuzzers.

(3) The third challenge Muench et al.[23] mentions is instrumentation. Instrumentation on desktop systems is used to obtain coverage information about the software that is being fuzzed and detect memory corruptions by adding them during compile or run time. Instrumentation being added during compile time therefore requires the firmware beforehand. This is already an issue on IoT devices, since acquiring the firmware is not always possible. Additionally, the variety of operating systems and processor architectures, makes instrumentation on IoT devices a challenging task. Moreover, obtaining the manufacturer's tool chain to re-compile the firmware with instrumentation is rarely possible. A workaround to this approach could be the use of binary dynamic instrumentation frameworks like valgrind[34] or using QEMU's instrumentation[33], but these methods heavily depend on the OS and CPU architecture.

IV. Overview of IoT Tools and Techniques

Here we give an overview of different IoT fuzzers, their techniques and list their advantages and disadvantages (Table I).

A. Input Generation

1) Mutational fuzzing: Mutation based fuzzing is a method of input generation[25]. Mutational fuzzing requires a set of predefined messages to start the mutation on. These mutations can include e.g. bit flipping, checking for out of bound bugs, sending empty data or substituting parts of the message with random data[11] to explore new program states or trigger unexpected behaviour. This way the fuzzing process can get started easily with only a couple of, so called, seed messages. A disadvantage of mutational fuzzing is the

Tool	Technique	Target	Fuzzing Techniques	Crash detection
SIoTFuzzer[37]	Black box	Web Interface	Stateful Message Generation*	Network Monitor
IoTFuzzer[6]	Black box	Companion App*	Generation Mutation Taint analysis	Passive probing
Firm-AFL[39]	Grey box	Firmware	Mutation Augmented Process Emulation*	Emulation
Snipuzz[11]	Black box	API	Snippet-based mutation*	Network Monitor
Firmcorn[13]	Grey box	Firmware	Optimal virtual execution* Vulnerability-oriented fuzzing*	Instrumentation
Diane[29]	Black box	Companion App	Under-constraint Input Generation*	Passive probing Active Probing
HFuzz[20]	Grey box	Network protocol	Message Structure Tree*	Instrumentation
WMIFuzzer[35]	Black box	Web Interface	Mutation Generation	Network Monitor

* = Novel technique in fuzzer

Table I
An overview of different IoT fuzzing tools.

limited coverage. A mutational fuzzer can rarely generate input, that deals with a target's complex sanity checks, since mutational input generation does not take the input format into account[25].

2) Generational fuzzer: Generation based fuzzers create messages from scratch while being provided with the format specifications for the input. Creating such a format specification requires manual effort and may even be infeasible, especially if the format is not available[25]. In the work of Srivastava et al.[32] they attribute the performance of FirmFuzz to their generational approach of input generation, due to resulting the constrained state space, that leads to a decreased overhead.

3) Under-constrained Input Generation: Under-constrained Input Generation is a technique utilized by the fuzzer DIANE. Here a combination of static and dynamic analysis is used on the companion app to find functions, that produce "valid yet under-constrained" inputs for the IoT device. The companion app's own functions are then used to generate input for the IoT device, that is not constrained by the app and structurally correct enough to not be discarded by the IoT device. The limitations of this approach lie in the implementation of the app analysis to find the desired functions. Additionally, since this is a back box approach to input generation, coverage is another issue.

4) Snippet-based mutation: Snipped-based mutation is a novel approach to input generation of Snipuzz[11]. Snippet-based mutation is the application of the mutation-based fuzzing approach on snippets. Snippets are parts of messages, determined by a heuristic algorithm and hierarchical clustering. Those snippets are categorized by the response they trigger from the IoT device. Snippets are then used to build new messages to trigger new program states. This method of mutation and message generation creates messages, which more likely follow message or protocol constraints of IoT devices, which leads to more effective fuzzing. Since this mutation method is guided by the response of the tested device, detailed responses are required to accurately categorize snippets[11].

5) Message Structure Tree: Message Structure Tree is a mutational fuzzing technique where the valid input is analysed to create a tree structure based on heuristics to mutate single fields of the input[20]. This way, the grammar of the protocol can be derived without explicitly providing the input format.

6) Stateful Message Generation: This technique was introduced by SIoTFuzzer[37] which fuzzes web interfaces of IoT devices. Stateful Message Generation (SMG) is divided into three parts: front-end analysis, state analysis and seed generation. SMG considers that communication depends on certain states and therefore groups together messages as a test case to fuzz the target system. So far, SMG is only used to fuzz web interfaces in SIoTFuzzer[37].

B. Instrumentation

1) Binary Rewriting/Instrumentation: Binary rewriting can be used to add instrumentation to firmware[23]. Instrumentation can be used to, e.g. add hooks to specific functions. This is interesting for fuzzing once instrumentation is added to internal exceptions to check for crashes or otherwise unexpected behaviour[13]. To perform binary rewriting, disassembly of the firmware is necessary, which requires partial decompilation. An additional challenge is the missing room for additional instrumentation due to embedded devices being optimized for their memory usage[23].

2) Symbolic Execution: A technique used to increase code coverage by using symbols as input and tracking manipulations and comparisons of them during runtime[36]. The usage of the input is then backtracked to solve the constraints of specific code branches if a desired state is reached. Symbolic execution has the problem of path explosions and constraint solving, which poses as an obstacle to scalability[7]. Path explosions is the exponential increase of code branches the larger the program is. A part of this problem are possible infinite loops. Constraint solving can pose another challenge, since depending on "how deep" the program's tracking goes, the calculation of the constraints of a specific branch can be very complex[28].

3) Taint analysis: Taint analysis is used to track data of interest during execution. The data that is being tracked is called taint source. IoTFuzzer[6] uses taint analysis to track, e.g. user input to find out which input influences network messages sent to the analysed IoT device.

C. Emulation

1) Full Emulation: Emulation tackles the problems of throughput and scalability in IoT fuzzing. This is done by improving the performance, success rate and hardware indipendence of fuzzers[11]. Full emulation of the firmware, with the help of heuristics, mitigates the lack of fault detection and increases accuracy of found vulnerabilities to a level of desktop system application fuzzers. Additionally, emulation based fuzzing provides the possibility to repeat test cases and their executions to further analyse specific test runs[24]. Often third party developers lack details of the device to implement a good emulator. This makes building emulator require huge amounts of manual effort[23], due to IoT devices heavily dependence on their hardware[39]. Failing to emulate even a part of a device or its peripherals may lead to the firmware not running at all[32].

2) Partial Emulation: Partial emulation can lead to accurate vulnerability detection with decreased performance in comparison to full emulation but possibly better performance than fuzzing the physical device, since it makes the fuzzing process more scalable[23]. Partial emulation is done by only emulating parts of the firmware or its peripheral devices.

3) Augmented Process Emulation: This method of emulation is proposed and used by Firm-AFL[39]. Augmented process emulation utilizes both system-level emulation and user-mode emulation to increase execution speed of the tested firmware or software. Here system-level emulation is only used when necessary, due to its low speed, while user-mode emulation is used the rest of the time. This improves the overall throughput of fuzzers utilizing Augmented Process Emulation compared to fuzzers using emulators, that only make use of system emulation. Currently, Augmented Process Emulation is limited to firmware that can be emulated in a system emulator and runs a POSIX-compatible operating system.

4) Optimized Virtual Execution: This technique used by Firmcorn[13], where the firmware instructions are executed in a lightweight CPU

emulator. This approach circumvents the overhead generated by full-system emulation. The execution is further optimized by using heuristic algorithms like omitting unnecessary functions to optimize the execution process.

D. Code Coverage

1) Vulnerability-oriented fuzzing: Vulnerability-oriented fuzzing is used in Firmcorn[13]. For this method, static analysis is used to find vulnerable code. Vulnerable code is determined by multiple factors like, complexity, number of memory operations and call to sensitive functions. Those attributes are calculated based on information about the target's control flow, like the number of edges of a function or the cyclomatic complexity of a function.

2) Coverage-oriented fuzzing: Coverage-oriented fuzzing generates input with the traversion of different execution paths in mind. This is done to maximize code coverage to reach paths which may be vulnerable by taking the ability of an input to trigger new paths into account[28]. While coverage guided fuzzing tries to maximize code coverage, usually most of a software's code is not vulnerable, therefore a lot of resources are spent on exploring invulnerable code paths.

3) Directed fuzzing: Direct fuzzing is the process of generating input with the goal of traversing specific execution paths[28]. Since only a fraction of firmware code has vulnerabilities, the grey box approach to fuzzing by only focusing on code coverage leads to many test cases, that end up not finding vulnerabilities[13].

E. Crash Detection

1) Active Probing: Active probing is used to determine the state of the target by regularly sending messages to the target. The response of the target to such a message is known. Should the response deviate from the expected message or should the device not respond at all, it can be assumed that there is an error.

While this probing method can detect errors that do not lead to crashes, the probing messages could lead to unexpected states of the target themselves. Sending additional messages to probe for the liveness of the target, also slows down the overall fuzzing process, since such probing messages do not contribute to increasing the coverage of the target.

2) Passive Probing: During passive probing the messages, that are sent for fuzz testing, are used to determine the state of the target. While the target device responds in an acceptable time window, it assumed, that no crash has occurred.

V. Related Work

A. Static Analysis

Alternatively to fuzzing, there are other ways to test software for vulnerabilities such as static firmware analysis. Static firmware analysis is the analysis of firmware without executing it by using tools like binwalk[4] to unpack the firmware and reverse engineering it with a reverse engineering tool like IDA[17][10]. The advantage of static analysis is the possibility to automate and scale the processes of analysing the firmware[8], since the testing does not depend on a physical device. On the other hand, static analysis also yields a high amount of false positives and may not find completely new vulnerabilities with the usage of its heuristics[39]. Another challenge during static analysis is the handling of packed or obfuscated code, since it first has to be unpacked or deobfuscated to perform meaningful analysis on it[8].

1) Dynamic Analysis: Dynamic Firmware analysis is another alternative to fuzzing. For dynamic analysis, the firmware is executed to be investigated. This can be done in a multitude of ways. For example, by running the firmware on the original device or emulating the device to have the firmware run in a virtual environment. The running firmware's behaviour is then analysed[9]. The challenge of working with packed or obfuscated code during static firmware analysis can be overcome with dynamic analysis[36] by emulating the physical device, which increases scalability and eliminates the need to acquire the physical device to test it[9].

VI. Conclusion

In this paper we created an overview of the different IoT fuzzing techniques used by state-

of-the-art IoT fuzzing tools and compared their approaches in regard to input generation, crash detection and their device scopes. The IoT fuzzers we looked at, utilized many techniques to make use of many attack surfaces and even used software outside the IoT devices themselves to gain information about the device, like IoTFuzzer[6]. There were also fuzzers, which did not create a new approach to fuzzing itself, but applied existing fuzzing techniques to the field of IoT fuzzing.

All in all, there are many techniques used in the field of IoT fuzzing, including some that are even outside the field of conventional fuzzing, such as symbolic execution, which belongs more in the class of dynamic analysis techniques. This makes fuzzing a very diverse topic for research, in which there is a lot of room for improvement.

References

[1] *american fuzzy lob*. https://github.com/google/AFL.

[2] *American Fuzzy Lop plus plus*. https://github.com/AFLplusplus/AFLplusplus.

[3] Manos Antonakakis et al. "Understanding the Mirai Botnet". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[4] *Binwalk*. https://github.com/ReFirmLabs/binwalk.

[5] *boofuzz*. https://github.com/jtpereyda/boofuzz.

[6] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing". In: *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_01A-1%5C_Chen%5C_paper.pdf.

[7] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046. URL: https://doi.org/10.1109/SP.2018.00046.

[8] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. "A Large-Scale Analysis of the Security of Embedded Firmwares". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, 2014, pp. 95–110. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin.

[9] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*. Ed. by Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang. ACM, 2016, pp. 437–448. DOI: 10.1145/2897845.2897900. URL: https://doi.org/10.1145/2897845.2897900.

[10] Yaniv David, Nimrod Partush, and Eran Yahav. "FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. Ed. by Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar. ACM, 2018, pp. 392–404. DOI: 10.1145/3173162.3177157. URL: https://doi.org/10.1145/3173162.3177157.

[11] Xiaotao Feng et al. "Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference". In: *CoRR* abs/2105.05445 (2021). arXiv: 2105.05445. URL: https://arxiv.org/abs/2105.05445.

[12] The OWASP Foundation. *Fuzzing | OWASP*. 2021. URL: https://web.archive.org/web/20210414111843/https://owasp.org/www-community/Fuzzing (visited on 04/14/2021).

[13] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. "FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution". In: *IEEE Access* 8 (2020), pp. 29826–29841. DOI: 10.1109/ACCESS.2020.2973043. URL: https://doi.org/10.1109/ACCESS.2020.2973043.

[14] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. "Operating Systems for Low-End Devices in the Internet of Things: A Survey". In: *IEEE Internet Things J.* 3.5 (2016), pp. 720–734. DOI: 10.1109/JIOT.2015.2505901. URL: https://doi.org/10.1109/JIOT.2015.2505901.

[15] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Grammarinator: a grammar-based open source fuzzer". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*. Ed. by Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir. ACM, 2018, pp. 45–48. DOI: 10.1145/3278186.3278193. URL: https://doi.org/10.1145/3278186.3278193.

[16] Mark Hung. "Leading the IoT Gartner Insight on How to Lead in a Cnnected World". In: *Gartner Research* 1 (2017), pp. 1–5.

[17] *IDA Pro*. https://hex-rays.com/ida-pro/.

[18] Mohd Ehmer Khan, Farmeena Khan, et al. "A comparative study of white box, black box and grey box testing techniques". In: *Int. J. Adv. Comput. Sci. Appl* 3.6 (2012).

[19] Eduard Kovacs. *Over 500,000 IoT Devices Vulnerable to Mirai Botnet*.

2016. URL: https://web.archive.org/web/20210507170030/https://www.securityweek.com/over-500000-iot-devices-vulnerable-mirai-botnet (visited on 05/07/2021).

[20] Xinyao Liu, Baojiang Cui, Junsong Fu, and Jinxin Ma. "HFuzz: Towards automatic fuzzing testing of NB-IoT core network protocols implementations". In: *Future Gener. Comput. Syst.* 108 (2020), pp. 390–400. DOI: 10.1016/j.future.2019.12.032. URL: https://doi.org/10.1016/j.future.2019.12.032.

[21] Valentin Jean Marie Manès et al. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. DOI: 10.1109/TSE.2019.2946563.

[22] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. *Fuzzing: the state of the art.* Tech. rep. DEFENCE SCIENCE and TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[23] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices". In: *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society, 2018. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_01A-4%5C_Muench%5C_paper.pdf.

[24] *Panda.* https://github.com/panda-re/panda.

[25] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: Fuzzing by Program Transformation". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA.* IEEE Computer Society, 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056. URL: https://doi.org/10.1109/SP.2018.00056.

[26] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "Model-Based Whitebox Fuzzing for Program Binaries". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 543–553. ISBN: 9781450338455. DOI: 10.1145/2970276.2970316. URL: https://doi.org/10.1145/2970276.2970316.

[27] Brien Posey. *IoT devices.* 2021. URL: https://web.archive.org/web/20210520072243/https://internetofthingsagenda.techtarget.com/definition/IoT-device (visited on 05/20/2021).

[28] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing". In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* The Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

[29] Nilo Redini et al. "DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices". In: *42nd IEEE Symposium on Security and Privacy 2021.* 2021.

[30] Nilo Redini et al. "Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 2020, pp. 1544–1561. DOI: 10.1109/SP40000.2020.00036. URL: https://doi.org/10.1109/SP40000.2020.00036.

[31] Nilo Redini et al. "Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 2020, pp. 1544–1561. DOI: 10.1109/SP40000.2020.00036. URL: https://doi.org/10.1109/SP40000.2020.00036.

[32] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard E. Shrobe, and Mathias Payer. "FirmFuzz: Automated IoT Firmware Introspection and Analysis". In: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, IoT S&P@CCS 2019, London, UK, November 15, 2019.* Ed. by Peng Liu and Yuqing Zhang. ACM, 2019, pp. 15–21. DOI: 10.1145/3338507.3358616. URL: https://doi.org/10.1145/3338507.3358616.

[33] *TriforceAFL.* https://github.com/nccgroup/TriforceAFL.

[34] *Valgrind.* https://www.valgrind.org/.

[35] Dong Wang, Xiaosong Zhang, Ting Chen, and Jingwei Li. "Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface". In: *Secur. Commun. Networks* 2019 (2019), 5076324:1–5076324:19. DOI: 10.1155/2019/5076324. URL: https://doi.org/10.1155/2019/5076324.

[36] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares". In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014.* The Internet Society, 2014. URL: https://www.ndss-symposium.org/ndss2014/avatar-framework-support-dynamic-security-analysis-embedded-systems-firmwares.

[37] Hangwei Zhang, Kai Lu, Xu Zhou, Qidi Yin, Pengfei Wang, and Tai Yue. "SIoTFuzzer: Fuzzing Web Interface in IoT Firmware via Stateful Message Generation". In: *Applied Sciences* 11.7 (2021), p. 3120.

[38] Nan Zhang et al. "Understanding IoT Security Through the Data Crystal Ball: Where We Are Now and Where We Are Going to Be". In: *CoRR* abs/1703.09809 (2017). arXiv: 1703.09809. URL: http://arxiv.org/abs/1703.09809.

[39] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.*

Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 1099–1114. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

VII. Appendix

A. Reconnaissance

To gain information about the system, we start off with *nmap*. The result of the port scan resulted in 6 open ports: 22 (ssh), 53 (dns), 80 (http), 443 (https), 5515 (unknown) and 65534 (unknown). Knowing that there was a backdoor service on this device, it was probably on either port 5515 or 65534, since those are not part of the IANA well-known ports.

```
0  nmap $TARGET -p-
1  nmap $TARGET -sV -sC -p22,53,80,443,5515, 65534
2
```

Figure 3. SYN scan all ports and detailed scan over open ports.

B. Getting shell and adding user

Connecting with port 5515 via *netcat* returned a root user shell. To add a user, I simply edited the */etc/shadow* and */etc/passwd* file by adding one entry in each file. The entry for the */etc/passwd*-file contained the username, uid etc. and the other one contained the username, md5 hashed password etc. for the */etc/shadow*-file. To

```
0  echo "echo tuan:x:1001:1001::/root:/bin/ash >> /
    etc/passwd;exit" | nc -nv $TARGET 5515
1  echo 'echo tuan:\$1\$123456\
    $qqQvjw0PqIk7otmzNsUIN0:18145:0:99999:7::: >> /
    etc/shadow;exit' | nc -nv $TARGET 5515
2
```

Figure 4. Adding user "tuan" with the password "password".

check whether the user was added correctly, I logged in via SSH with the new user.

C. SSH Brute-force

For brute forcing the SSH login for "iotgoatuser" I used *hydra*. An alternative to brute forcing over ssh would be getting the */etc/shadow* and */etc/passwd* files and cracking the passwords of its users locally with tools like *JohnTheRipper* or *Hashcat*. This method would circumvent defence mechanisms like *fail2ban*, although the usage of such defence mechanisms is unlikely on an IoT device.

```
0  hydra -l iotgoatuser -P ./data/passwords.txt ssh
   ://TARGET -t 4 -f
1
```

Figure 5. Brute-force ssh

D. MITM

When visiting the web-interface of the IoT device via Firefox, we are greeted with a warning, that the certificate is self-signed. This poses a threat to the user, since self-signed certificates can not be revoked and don't expire. If the certificate was somehow leaked, the integrity of the website could not be restored without replacing the certificate, which may not be easily done on an IoT device sold to hundreds or thousands of consumers. Self-signed certificates are used nonetheless on IoT devices, since they are easier to obtain and free of charge.

To proceed with the testing, I had to simply press "Accept the Risk and Continue" in the browser.

Logging in on the web interface with the credentials we obtained in the brute forcing step didn't seem to have worked. I then tried brute forcing the login form with burpsuites "Intruder" function and a wordlist from SecLists[1], using the root-user and the iotgoatuser-user, which didn't work either.

Using the backdoor, I then changed the root password to "asdfasdf", since the already existing password didn't seem easily crackable. Logging in with the new credentials worked.

Looking at the *luci*-directory in */usr/lib/lua/luci/* we found */usr/lib/lua/luci/controller/iotgoat/iotgoat.lua*, which lists the secret developer page under *https://$TARGET/cgi-bin/luci/admin/iotgoat/cmdinject*.

E. Static analysis

To start the static analysis, we first extract and unpack the filesystem by finding the filesystem in

[1]https://github.com/danielmiessler/SecLists

the firmware with *binwalk*, extracting it with *dd* and unpacking it with *unsquashfs*. This gives us access to the whole file system of the IoT device we are analysing.

```
0   binwalk ./data/Syssec\ IoT\ Device.bin
1   dd if=data/Syssec\ IoT\ Device.bin of=data/0
    x1F5A50 bs=1 skip=2054736 count=2813038
2   unsquashfs data/0x1F5A50
3
```

Figure 6. Extracting and unpacking filesystem

To find the shadow and passwd file, we can simply run a *find* command to look for them or, by simply knowing, that they are usually in the */etc/* directory.

The same can be done for the certificate to find a certificate in */etc/ssl/certs/ca-certificates.crt*.

F. Write-up

The full write-up can be found here: https://git.uni-due.de/sktatran/syssec-embedded-security-writeup/-/blob/main/writeup.org