Overview of IoT Fuzzing Techniques

Tuan-Dat Tran
(3012345)

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 on the security of those devices. To address this issue, there have been many approaches in detecting, analyzing and mitigating security flaws in IoT devices. One of the ideas to detect vulnerabilities in an automated manner is IoT Fuzzing. Contrary to regular fuzzing it comes with its own constraints and techniques to optimize performance and coverage of attack surfaces.**

In this paper we are comparing techniques used by IoT fuzzers to circumvent the adversities presented by IoT devices like app-based approaches by IoTFuzzer and Snipuzz or emulation approaches used by Firm-Afl.

Due to the wide range of different IoT fuzzing tools we are dividing the comparison of the techniques based on the type of IoT fuzzing tool. We also outline the evolution of IoT fuzzing techniques to visualize the progress made in the field. This overview can then be used to choose the optimal usage of a specific IoT fuzzing device in a given use case or combine different techniques used in different fuzzing tools to create a novel approach and find new security flaws through a combined usage of IoT fuzzing techniques.

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[18]. 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". This led to over 70 unique attack incidents[21] between 2010 and 2016 while the number of IoT devices and embedded systems in use is steadily rising and with it the amount of vulnerabilities in the wild.

While implementation flaws and app overprivilege 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[22]. One approach to discover those flaws is called fuzz-testing, or fuzzing. Mitigation of found security flaws can often be hard due to the nature of embedded devices being heavily customized and often not adhering to one specific standard. Therefore, the fixing of security flaws is often left to the manufacturer of the device, since they possess the necessary tool chains, source code and pipelines to provide security patches to their devices.

Fuzzing is a method to test software for flaws by automatically generating and sending malformed data to the software. There are many ways to generate and send data to the software. An example for a specific type of input generation is mutation based fuzzing, which is utilized by IoTFuzzer[6][13]. Mutation based fuzzing takes a valid input and changes specific parts of it to trigger an unexpected state in the software and therefore crash it. Crashing or bringing the software into an unexpected state is the general goal of fuzzing, since behavior like this indicates the presence of a bug.

Due to fuzzing being an automated process, fuzzing became a common tool for software testing in software development. Conventional fuzzing of software can be easily done concurrently, since software can, in most cases, be easily executed concurrently[22]. This increases the throughput of the fuzzer and thus the amount of test cases the software is tested against. This is one of the issues, which IoT fuzzers have to

deal with, since the fuzzing IoT devices usually includes fuzzing the physical device itself if there is no emulation solution available. While emulation increases scalability, it also enables another range of issues and complexity to the fuzzing process e.g. the acquisition of the firmware to emulate. The process of firmware acquisition is different for every device, since it is dependent on the willingness of the manufacturer to publicly release the firmware. If the manufacturer does not release the firmware for his device, the firmware needs to be extracted directly from the device, which can vary in difficulty depending on the device[22].

Alternatively to fuzzing there are other ways to test software for vulnerabilities like static and dynamic 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 engineer it with a reverse engineering tool like IDA[19][10]. For dynamic analysis the firmware is executed to be investigated. This can be done in a multitude of ways, for example running the firmware on the original device or emulating the device to have the firmware run in the emulated environment. The running firmware's behavior is then analyzed[9]. The advantage of static analysis is the possibility to automate and scale the processes of analyzing 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. This can be overcome with dynamic analysis[37], by emulating the physical device, which increases scalability and eliminates the need to acquire the physical device to test it[9].

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 fuzzers can also make use of techniques used by dynamic analysis since both approaches require execution of the firmware. This makes emulation a feasible way of testing IoT devices to increase scalability[15]. In this work we will focus mainly on fuzzers, which were primarily developed for IoT fuzzing.

Even though IoT fuzzers are used for finding security vulnerabilities in devices, and fixing those errors or learning from them and mitigating them is the next logical step, we will not discuss mitigation techniques in this paper since this is outside of our scope. We will also not dive deep into the implementations of specific techniques.

By creating an overview of different IoT fuzzing techniques, we hope to archive a comprehensive list of IoT fuzzing tools and their properties to help developers and researchers to find the right tool for their job and weigh in the positive and negative aspects of existing approaches.

II. Background

A. IoT devices and embedded systems

The terms IoT devices and embedded systems describe a great amount of devices. Embedded systems are devices which interact with their surroundings via sensors and regulators and are built to serve a specific purpose[22]. IoT devices on the other hand are broadly described as devices which extend regular devices with an internet connection and enable them to communicate over it[26]. 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, which includes the definition of internet capable embedded systems. Ongoing, when we describe IoT devices, the description also fits embedded systems if not explicitly mentioned otherwise.

The wide applicability of IoT devices in the context of business, manufacturing and home-use increases the surface area for vulnerabilities to be found. IoT devices, being so diverse regarding their functionalities and ways to offer their services, further increases the possible ways to accumulate vulnerabilities.

IoT devices, due to being built for specific purposes, don't need as much processing power as a general computer does. This leads to them having a hardware platform specifically tailored

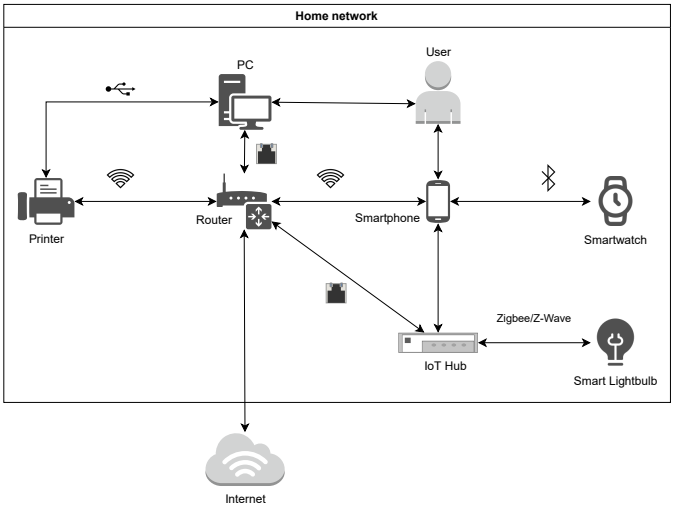

Figure 1. Example of IoT home network (inspred by Wang et al.[35]).

to their use case. And due to the heterogenic nature of IoT devices in terms of e.g. OS, 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 capability or memory footprint[16].

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

IoT firmware is the bridge between the hardware of the device and the software running on it. Sometimes IoT firmware can be acquired through the vendors website. Alternative methods for acquiring the firmware are extraction from the physical device, even though this way can be challenging due to debugging ports (e.g. JTAG interface) to dump the firmware from the device may not be available[6][30]. Firmware running on an IoT device expects the presence of certain hardware at boot- and/or runtime. Therefore missing hardware may cause the device to get stuck in a busy loop trying to find the hardware[32]. Additionally firmware is often packed or even encrypted, which poses as an obstacle for firmware analysis. In some cases proprietary compression algorithms or encryption without knowledge of the secret key makes firmware analysis infeasible or even impossible.

The works of Hahm et al.[16] 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. In the works of Muench et al.[22] a similar classification is used. They 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 OS (operating system) is often modified to be more lightweight and offer a lightweight user environment like busybox. T2 (Type-2) devices run on customized operating systems which are tailored to the devices use case. In order to save space and computational power, typical OS 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 devices functionalities, are a single instance. This leads to a so-called “blob firmware”[31], consisting of the application and system code compiled together. Muench et al.[22] add that the classification of the device merely indicates the kind of available security mechanisms while the usage of them varies from device to device.

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 tasks they were built for. Therefore functionalities like heap hardening may not be present due

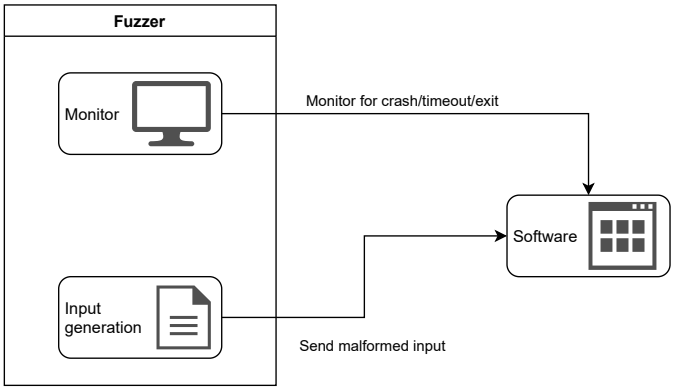

Figure 2. Generalization of fuzzing process.

```
0   > echo "rm -rf / --no-preserve-root" | radamsa -n
     5
1   rm -rf / --no-presef / --no-preserve-root
2   rm -rf / --no-preserve-root
3   rm -rf  --no-preserve-roo
4   rm -rf!!;xcalc\0\u0000&#000;\n
     \340282366920938463463374607431768211457!xcalc$'%
     d\0$!!%d\x00 / --no-preserve-root
5   rm -rf / --no-preserve-r'xcalc%#x'xcalcaaaa%d%n\0\
     x0aNaN%#x%p%d;xcalc+infoot
6
```

Figure 3. Example output of radamsa on "rm -rf / –no-preserve-root" (omitted non-printable characters)

to the IoT devices limited computing power and constrained costs[22].

B. *Fuzzing*

Fuzzing describes the process of testing a software for faulty and unexpected behavior by sending malformed messages as input for the software[13]. The basic fuzzing process can be divided into three steps: (1) input generation (2) sending messages as input to software and (3) monitor software behavior in reaction to the given input (Figure 2). Due to the need to have the tested software running, fuzzing is considered a dynamic technique. 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[22].

There are multiple types of fuzzing techniques based on the amount of known information about the software: Whitebox, blackbox and greybox fuzzing. Whitebox fuzzing has complete information about the software's source code. Blackbox fuzzing on the other hand has no such information while greybox fuzzing lies in between regarding the available information. Blackbox fuzzing relies purely on the binary of a program or the program in its already executed state[20]. This leads to blackbox fuzzers generally creating many unnecessary test cases due to the lack of knowledge about the target[15]. Greybox fuzzers may use the additional information to improve the monitoring by injecting instrumentation to the binary at compile time[7]. Whitebox fuzzers can utilize the full source code to increase efficiency by using techniques like symbolic execution or dynamic taint analysis[28]. Comparing blackbox fuzzers with greybox fuzzers or even whitebox fuzzers is therefore not worthwhile, due to the different starting conditions[11].

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 fuzzed. The fuzzer radamsa[27], a general purpose blackbox fuzzer, for example creates messages derived from a possibly valid input and changes parts of it to generate new test cases. This classifies it as a mutation based fuzzer, since radamsa modifies existing input to create test cases. The operations on the given input can be substitution of characters, bit flips or other operations, based on the tools internal heuristics (Figure 3). There are lists, which contain strings that have a high probability to cause issues when used as input[3][14]. These lists may be used by fuzzers as well to generate input but the generated input can also be random. The goal is to find an input which makes the software crash or display otherwise unexpected behavior.

The message sending step depends on the target of the message. Software offers many ways to interact with it, from simple things like user input via text fields in desktop applications to packages

sent by the users through web browsers to web servers. Those points of contact are possible targets for fuzzing. And dependent on the target, different techniques for message generation may be used. If a network protocol is fuzzed, like with the tools boofuzz[5], the fuzzer needs to have an understanding of the network protocol which is fuzzed. While other fuzzers like XSStrike[36], which was built to find XSS (cross site scripting) bugs, target web applications. While XSS bugs will not crash the software, they are a serious security threat, which enable an attacker to inject code to websites[12].

Monitoring the softwares behavior upon receiving a malformed message as input is the last step of a typical fuzzing loop. The behavior monitored depends on the earlier steps, but after every loop the original programs 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 don't try to trigger crashes usually use the applications answer to the input message to determine whether the test case triggered the event, which was tested for[5]. To monitor a softwares crash the fuzzer can provide instrumentation, with which the tested software is compiled[1], if greybox fuzzing is used. A blackbox 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 greybox or whitebox 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 aren't 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 don't 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 softwares protocol[5], grammar[17] or model[25]. 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.

III. Intricacies of IoT Fuzzing

IoT Fuzzing is the application of fuzzers on IoT devices, which poses new challenges, since fuzzing hardware and their firmware and fuzzing software work on different domains, each with their own challenges.

The works of Muench et al.[22] offers insight into the challenges of fuzzing embedded devices. They mention three main challenges. The first challenge being fault detection. Regular fuzzing assumes that crashes are generally observable. Due to an IoT devices limited computational capability fault detection functionalities, usually present in multi-purpose devices, are rarely present in embedded systems. Even when crash causing fault detection mechanisms are available, they would be logged on multi-purpose systems while embedded devices usually do not provide feedback like multi-purpose systems do due to the lack the necessary I/O capabilities. A liveness check, also called probing, can be performed to check the status of the device while fuzzing it. Probing can be either active and 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 devices responses to the test message to determine liveness or observes visible crashes.

Muench et al.[22] expands on this by classifying system crashes by their observability. An observ-

able crash is therefore the most visible and managable kind of crash, where the tested device stops running and provides an error message or another 6 that is easily visible. It is added that this also includes crashes, which don't provide additional information about the crash. Observable crashes are the optimal case regarding crashes, since they are visible and enable the fuzzer or tester to react without delay. Reboots are another kind of crash. Crashes on T3 devices automatically lead to a reboot, since the crashed software and firmware on the device are part of the same "blob firmware". On other kinds of devices, a service may crash while the rest of the system continues to run without problems. In reaction to malformed input a device may hang. That means that it halts execution and doesn't react to any more input. This may be due to being stuck in an infinite loop. This leads to a slowdown in throughput and the device needs to be restarted if such behavior is found. Late Crashes pose a challenge for testing the device. This behavior is described as the device crashing after a non-negligible amount of time after the cause of the crash, like a malformed message, is sent, which 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. This can lead to wrong data and incorrect results. This malfunctioning of the device is hard to detect, since the fuzzer needs information about the expected response to determine whether its an output caused by a malfunction or not. This is further complicated due to the diverse message formats in use[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].

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 with the same starting conditions. This is not a challenge with regular software, since the softwares original state is reestablished after rerunning it and changes on the file system can be reverted with a e.g. snapshot of the virtual VM (virtual machine). Restarting an IoT device can take up to a minute, since the device needs to be completely rebooted to get it to a neutral state.

The third challenge Muench et al.[22] mentions is the 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 time 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. Obtaining the manufacturers tool chain to re-compile the firmware is rarely possible. This could be solved by utilizing binary dynamic instrumentation frameworks like valgrind[34] or using QEMUs instrumentation[33], but these methods heavily depend on the OS and CPU architecture.

Furthermore, IoT fuzzing suffers from the similar or the same problems as regular fuzzing based on the fuzzing approach. Therefore an IoT fuzzer which utilizes network protocol fuzzing will face the same challenges as the used network protocol fuzzer, like generating valid input[29], on top of the aforementioned challenges of fuzzing an IoT device.

IV. Overview of IoT Tools and Techniques

In this section, we are going to create an overview of different IoT fuzzers, list the techniques they utilize and look at the techniques advantages and disadvantages (Table I).

A. Mutational fuzzing

Mutation based fuzzing is a method of input generation[24]. Mutational fuzzing requires predefined messages to start the mutation on. These mutation 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 behavior.

B. 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 a format is not available[24].

C. 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].

D. Message Structure Tree

Message Structure Tree is a mutational fuzzing technique where the valid input is analyzed 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.

E. Binary Rewriting/Instrumentation

Binary rewriting can be used to add instrumentation to firmware[22]. 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 behavior[15]. 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[22].

F. Full Emulation

Emulation tackles the problems of throughput and scalability in IoT fuzzing. This is done by improving the performance, success rate and hardware-indipendance 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 analyze specific test runs[23]. Often third party developers lack details of the device to implement good emulator. This makes building emulator requiring huge amounts of manual effort[22]. This is due to IoT devices being heavily dependent on their hardware[39].

G. 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[22]. Partial emulation is done by only emulating parts of the firmware or its peripheral devices.

H. 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

Tool	Technique	Target	Fuzzing Techniques	Crash detection
SIoTFuzzer[38]	Blackbox	Web Interface	Stateful Message Generation*	Network Monitor
IoTFuzzer[6]	Blackbox	Companion App*	Generation&Mutation Taint analysis	Passive probing
Firm-AFL[39]	Greybox	Firmware	Mutation Augmented Process Emulation*	Emulation
Snipuzz[11]	Blackbox	API	Snippet-based mutation*	Network Monitor
Firmcorn[15]	Greybox	Firmware	Optimal virtual execution* Vulnerability-oriented fuzzing*	Instrumentation
Diane[29]	Blackbox	Companion App	Mutation	Active probing
HFuzz[20]	Greybox	Network protocol	Message Structure Tree*	Instrumentation
WMIFuzzer[35]	Blackbox	Web Interface	Mutation	Network Monitor

* = Novel technique in fuzzer

Table I
An overview of different IoT fuzzing tools.

tested firmware/software. System-level emulation is only used when necessary, since it slows down execution. Currently augmented process emulation is limited to firmware that can be emulated in a system emulator and runs a POSIX-compatible operating system.

I. Optimized Virtual Execution

This technique used by Firmcorn[15] executes firmware instructions 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. Additionally the optimized virtual execution uses information about the context of the firmware.

J. Symbolic Execution

A technique used to increase code coverage by using symbols as input and tracking manipulations and comparisons of them during runtime[37]. 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 programs tracking goes, the calculation of the constraints of a specific branch can be complex.

K. Liveness Check

By checking for liveness a fuzzer can determine the state of an IoT device. This is done actively by sending regular heartbeat messages to the device or passively by checking for expected responses of the IoT device. Liveness checks may cause timeouts to be detected as crashes, which slows down fuzzing. Omitting active liveness check improves performance, since probing packages aren't sent, which make up a certain percentage of traffic that do not contribute to the detection of vulnerabilities during the fuzzing process.

L. 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 analyzed IoT device.

M. Stateful Message Generation

This technique was introduced by SIoTFuzzer[38] 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[38].

N. Vulnerability-oriented fuzzing

Vulnerability-oriented fuzzing is used in Firmcorn[15]. 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.

O. 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. This is done 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 softwares code is not vulnerable, therefore a lot of resources are spent on exploring paths, which are not vulnerable.

P. 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 graybox approach to fuzzing by increasing code coverage leads to test cases, which end up not finding vulnerabilities[15].

V. 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 regards to input generation, crash detection heuristics and their device scopes. The IoT fuzzer 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] which used the devices companion app to send fuzzing messages to the tested device.