Embedded Software Fuzzing: A Survey

Scottie Yu and Rumna Samanta

University of Waterloo, Waterloo, Canada

Abstract. Fuzzing in software testing has provided a new direction in finding vulnerabilities in the program. Since its discovery, it has become popular due to its simplicity and less overhead in deployment and its efficiency in finding real-world software bugs. To define briefly, fuzzing is an automated software testing method that inject a program with syntactically or semantically malformed inputs with an aim to discover software bugs in the system. A fuzzing tool runs a program with malformed input repeatedly and observe it for any exception like crashes or information leakage. These crashes reveals vulnerabilities or quality gaps of the system.

Fuzz-testing is proved to be quite effective in finding bugs in the normal software system. And, with then invention of Internet Of Things (IOT) embedded devices are becoming more and more essential to daily lives. Thus, it also important to ensure the security prospect of these devices. Fuzzing embedded devices to ensure its robustness would be an interesting topic to study. Thus, in this paper we have listed down the particularities about fuzzing and then surveyed on various challenges in implementing fuzzing on embedded devices, and on the existing solutions to these challenges and future direction about this topic.

Keywords: Fuzzing · Fuzz-testing · Program Under Test (PUT), · Fuzzer.

1 Introduction

The term Software Fuzzing was coined by Prof. Barton P. Miller[6] of University of Wisconsin-Madison in 1990. He was trying to feed a Unix program software with distorted program input, he found out that the inaccurate input data had caused the program to crash. He proposed that this strategy could be used to find vulnerabilities in a software program. He named this method evaluating the robustness of the program as "Fuzz-testing". Since then Fuzzing in Software testing has gained popularity because of its simplicity and effectiveness in discovering bugs in real world software system. For instance, google had launched ClusterFuzz project and use it to fuzz all the google product to find vulnerabilities in the system. As of May 2022, ClusterFuzz[7] has found 25,000+ bugs in Google (e.g. Chrome) and 36,000+ bugs in over 550 open source projects integrated with OSS-Fuzz (a free fuzzing platform for open source community).

Nowadays, attackers are also using fuzzing to unravel security vulnerabilities and for penetration testing. For example, in an interview, [8] the winner of the Pwn2Own challenge Charlie Miller talked about using automated fuzzing techniques to find crashes in a program. Several teams in the 2016 DARPA Cyber Grand Challenge (CGC) make use of fuzzing techniques in their cyber reasoning system. In order to defend their system, defenders are also employing fuzzing techniques to discover vulnerabilities in their program. For example, Cisco uses a mutational based fuzzer "The Mutiny Fuzzing Framework"[9] to fuzz network and discover bugs. Similarly, Google [10], Microsoft [11], Mozilla [12] uses fuzzing tools to find bugs in the system.

It is evident that fuzzing can be efficient method to ensure software reliability. Thus, more and more reputed companies are employing this method to ensure software quality. Apart from testing generic software systems, fuzzing can also be used in embedded systems. Recently, there has been many studies to find bugs in the embedded systems.

Network embedded systems are more ubiquitous these days. Electronic transactions now governs our daily lives, starting with accessing building with digital cards, payments for groceries, transportation payments, online shopping etc, everywhere we use embedded systems. With the advent of these electronic transactions, it also opened the door for transaction failures. These unexpected failures can cause disruption in daily lives. Thus, it is extremely important to find out bugs in the embedded system.

The embedded system can have vulnerabilities due the following reasons:

- **Design bug:** This type of bug could be introduced while transcription of specifications into functional specifications and then transcribed into technical specifications
- **Implementation bug:** Occurred while developing the program code.
- **Bug due to use:** This type of bug could be discovered while deployment or the operation of the program

Most of these bugs could be discovered and fixed during unit testing, integration testing and user acceptance testing. But, there some situation where some bugs could be completely unreported because of lack of proper resources in the testing.

For instance, vulnerabilities in the widely used contact-less card The MIFARE Classic had been exploited in the paper [13]. It highlighted a weakness in the pseudo-random generator of the card that acts a encrypted medium between the card and the player. Security of these embedded devices are crucial and the issues like Malware like Mirai [14] are highlighting these issues more. To resolve these issues and overcome the difficulty of conducting extensive testing, fuzzing or fuzz-testing could be employed to explore bugs in these embedded systems. And, it is particularly useful in large scale automation.

The common software systems have different mechanism to detect the defect states, embedded devices on the other hand lacks such mechanism to detect vulnerabilities. It is mainly due to the limitations of the I/O devices capabilities, constrained cost, and limited computing powers. Specifically, the silent memory corruptions on the embedded devices make the process of fuzzing the on the embedded system software significantly challenging. Unlike the generic software system, it is difficult to get the source code for the embedded system. Although, source code is not necessary to fuzz a system but it makes the process of fuzzing more efficient. For example, we can make use of compile time corruption detection techniques to detect anomalies in the software system. The embedded system lacks these as the source code is rarely available and there are lack of memory protection. Thus, it becomes significantly difficult to perform fuzz-testing on the embedded system.

In this paper we would highlight recent development on fuzzing a software system and our focus would be to discuss various methods that has been adopted recently to fuzz an embedded system particularly. We would also discuss various difficulties that embedded fuzzing faces and would also highlight possible solutions to those issues.

2 Fuzzing Definitions

After the advent of the method “Fuzzing” by Miller, it has been widely adopted in many areas of applications. For example, it is used in dynamic symbolic execution, function detection, robustness evaluation, kernel testing, complexity testing, representation dependence testing, penetration testing, embedded fuzzing, Fuzzing the Internet of Things (IOT) devices etc. To systematically represent the overall concept fuzzing we would define some basic terminologies of fuzzing in the following paragraphs.

- **Fuzzing:** It is an action of running of running a “Program Under Test” with randomly generated input data. Fuzz testing: The software testing method that utilise fuzzing mechanism to exploit vulnerabilities in a software system is termed as Fuzz testing
- **Fuzzer:** It is a program that performs the fuzz-testing on the Program Under Test
- **Seed:** A seed is a well – structured input to the PUT that is used to generate test cases by modifying it. Generally, fuzzers maintain a collection of seeds

and the collection may evolve over time. This collection of seeds is termed as seed pool.

- **Fuzz Campaign:** It is specific execution of a fuzzer on a PUT with specific security policy.
- **Bug oracle:** A bug oracle is a part of fuzz program which determines whether a given execution of the PUT violates a specific security policy
- **Fuzz Configuration:** It is basically the parameter values that controls the fuzz algorithm. The type of values in fuzz configuration depends on the type of the fuzz algorithm. If a fuzz algorithm sends a stream of random bytes to the PUT is said to have simple configuration space. Alternatively, more sophisticated fuzzers contain algorithms that accept a set of configurations and evolve the configuration over time. For example, CERT BFF [18] varies on the basis of mutation ratio and the seed over the course of a campaign.

3 Fuzzing taxonomy

In this paper we have divided the method of fuzzing into three categories based on how they execute a program, how they generate a test case and the techniques that they have employed for analysis. Method of program execution: We can subdivide the fuzzers into 3 categories based on how they execute a program. This categories are similar to software test execution. These are as follows

- **Black-box fuzzing:** This type of fuzzer make use of a fuzzing technique that do not have any prior knowledge of the internals of a Program Under Test (PUT). This method only observers input/output behavior of the PUT and treat it as a black-box. In the paper [19] Gopinath proposed an idea of BFuzzer which is based on black box fuzzing mechanism. It does not need any execution feedback such as code coverage from the target program, instead it depends on failure feedback. Meaning, BFuzzer checks whether the input to the program was valid or not. If the input was not valid then BFuzzer enumerates byte after byte of the program's input space until it find a valid "prefix". If the program does not find a valid "prefix" and returns "incomplete" then the program "reinitializes" and start testing again.

  In the paper [20], Kim et al proposed another idea of Black-box fuzzing using "smart seed selection" process. The authors of black-box fuzzing had argued that this method of fuzzing is more efficient comparing the grey and white box fuzzing. They also argued that in the case where source code of a program is not readily available or when the firmware is often heavily versioned, in such cases black-box fuzzing can be proved to be effective and efficient.

– **White-box fuzzing:** Contrary to the black-box fuzzing, white-box is a technique that generates test cases by analysing the internal code the Program Under Test (PUT). White-box fuzzing uses Dynamic Symbolic execution mechanism to find vulnerabilities in a program. In DSE, symbolic and Concrete execution operate concurrently, here concrete program states are used to simplify symbolic constraints, e.g., concertising system calls. White-box testing has also been used to describe fuzzers that employ tiant analysis. Because of this DSE implementations which may require dynamic instrumentation and implementation of SMT solvers, the white-box fuzzing is typically a programming overhead rather than black-box fuzzing. Although, white-box fuzzing includes overhead, the crash report generated by white-box fuzzing could exploit vulnerabilities like buffer-overflows, null-pointer differences and other index-out-of-bound bugs in a program.

– **Grey-box fuzzing:** Grey-box fuzzing takes midway approach comparing to black-box and white-box fuzzing. Grey-box fuzzing generally have some program information about the system under test. This type of fuzzing mechanism performs light weight static analysis on the program and gather dynamic information about its execution such as code coverage. An example of grey-box fuzzer is EFS [28] that uses code coverage gathered from each fuzz run to generate test cases with an evolution algorithm. Another example of grey-box fuzzer is BigFuzz [29] developed by Zhang. In this paper they had proposed a coverage-guided fuzz testing tool for big data analytics. This tool is useful for data-intensive scalable computing (DISC) applications. BigFuzz performs automated source to source transformations to construct an equivalent DISC application which is more amenable to fuzzing. Some modern fuzzers such as American fuzzy lop (AFL) [30] is an example of grey-box fuzzer. It is a security oriented fuzzing tool that employs compile-time instrumentation and genetic algorithms to generate clean and interesting test cases for a Program Under Test.

4 Types of embedded devices

The term ”Embedded System” can refer to a wide variety of systems and it is really difficult to clearly define what an embedded system is. Embedded systems are rapidly growing nowadays and they are integral part of the Internet of Things (IOT) devices. Embedded systems are designed to meet a special purpose and there could be other peripheral devices through which these devices interact with the physical world. Referring to the paper [31], we can divide the embedded devices into three categories based on the Operating System (OS) that they use. In this paper the authors selected type of OS as a distinguishable feature because it is responsible for handling the recovery state of a device from its faulty states and it may serve as the building block of more complex security primitives.

- **Type-I: General purpose OS-based devices** Inoder to suit embedded systems, sometimes general purpose operating systems are retrofitted into the embedded systems. For example, LINUX is widely used in various embedded systems.
- **Type-II: Embedded OS-based devices** In some devices where there is low computational power and the advanced Memory Management Unit (MMU) is not present , in such cases a custom operating system is used. uClinux, ZephyrOS or VxWorks are example of these system.
- **Type-III: Devices without an OS-Abstraction** These types of devices uses a "monolithic firmware" and its operation is based on a single control loop and interrupts which are triggered from the peripherals to handle events from the outer world.

We would be using these Type-I, Type-II and Type-III devices throughout the paper to denote the corresponding type of embedded devices.

5 Challenges with embedded fuzzing

Finding vulnerability specifically in embedded system is a challenge since the embedded system do not have any additional security. Fuzzing in embedded system could be an interesting option to exploit bugs as it can run from the outside of the system and it analyses the output of the PUT. Also, for some embedded systems with limited resources, it may freeze abruptly and it will stop responding. In such cases, the fuzzer of the system will not have any new data for the next input and eventually the effectiveness of the fuzzer will decrease with time. The issues with embedded fuzzing could be generalized into three main categories.

- **Fault Type:** This category includes all kinds of errors relating to memory and numerical computations. And, a wide range of fuzzing relies on crashes that can be observed immediately as a consequences of errors occurring a program execution. Generic software systems can have varied protection mechanism against this fault occurring while executing a program. But, these techniques are quite limited in the embedded systems and that cause a significant challenge to impose fuzzing for the embedded systems. Challenges relating to fault detection in the embedded system could be sub-divided into the following categories:
  - **Stack overflow:** In a situation where a program writes outside of the designated memory address, then this kind of error occurs. As a consequence a program may crash abruptly. While its easy to detect these kind of errors in software system, it is very difficult to detect in the embedded system with fewer resources. It may also happen that the error is detected only after when the program execution is stopped completely. So, for embedded systems if the memory is not corrupted then will not

have any discernible effect.

  - **Segmentation fault:** This type of error occurs when a program tries to access a specific region of the memory which is not allowed to access. This type of error occurs with the normal program, but in case of the embedded systems this violation has no effect since it is without OS.
  - **Memory corruption:** Memory corruption occurs when the memory is modified without an explicit assignment. On the embedded devices the problem of memory corruption is much less visible. The silent memory corruptions on the embedded system is habitual since it is a consequence of decrease on the effectiveness of the dynamic testing techniques.
  - **Numerical errors:** The typical errors like over/under-flow errors can occur in the system. Since the hardware components of embedded systems has limited computational power thus, it may cause these types of numerical error and subsequently memory corruption in the system.

– **Performance and scalability:** Since the desktop system has multi-processing feature, thus it is very easy to fuzz multiple systems in parallel in the normal software system. However, embedded devices has the limitation of resources and infrastructure, thus parallelization is a significant challenge in the embedded devices.

– **Response time:** It is the time elapsed from the reception of fuzz data and the time required to finally process the data and send it back. The response time for the embedded devices is large. Additionally, for efficient fuzzing it is required to restart and re-establish a clean state for generating next test case. It is quite easy to perform in desktop system, but it is very time consuming and difficult to achieve in embedded systems.

– **Waiting Time:** It is defined as the time elapsed between the request is launched to the PUT (Program Under Test) and the time when the response arrives. In case of embedded systems, it may also happen that the system had crushed during the process of fuzzing and couldn't provide any response to the system. In such case, the fuzzer decides whether to send another test case to stop testing because of a crash or to wait until the fuzzing process completes since it may be time consuming as well. So, in case of embedded systems it is necessary to increase the waiting time of the fuzzer and it may cause issues in the system.

– **Physical response:** How systems are responding physically to different situation while fuzz testing is an important measurement factor. When the volume and the rate of fuzz data increases, the temperature of the embedded devices also increases at these devices have a tendency to get warmer at high load. Thus, it is important to check the temperature of the devices while fuzz testing and make sure that the temperature does not reach to an undesired value.

- **Instrumentation challenges:** For efficient fuzzing, it is quite essential to get the state of the Program Under Test. Thus, normal desktop systems make use of compile-time and run-time instrumentation in order to collect code coverage information about the provided inputs and detect subtle corruption of the memory system. American Fuzzy Lop (AFL) is the most popular fuzzer that uses source code instrumentation to collect information about retrieved code coverage. While these components are readily available for the desktop system, these are not typically available for the embedded devices. Sometimes embedded devices consist of various peripheral devices which may have its own operating system and processor – and it is difficult to have a complete toolchain which may be used to fuzz these devices. Also, it is difficult to get access to the source code for the embedded devices to perform fuzz-testing on it. So, one solution to this problems could be to use binary dynamic instrumentation framework as proposed in Pin [32], Valgrind [33] and DynamoRio [34]. But these solution of using static and dynamic instrumentation tools is not available for embedded devices, thus instrumentation is big problem for embedded devices.

6 Survey of existing solutions

Pham et al. introduced a fuzzing program called Model-based Whitebox Fuzzing [1]. Model-based Whitebox fuzzing is designed to expore the functional sections of programs where valid inputs may uncover deep bugs that may occur during real execution of the program. To carry out this exploration, the authors use metadata about the file format and data chunks of known valid files to quickly get through the program's parser. Since this approach requires access to the source code, this project may not be applicable to scenarios where the fuzz testing is not being done by the developers of the system since many embedded systems manufacterers may not provide the source code of the system's firmware. Furthermore, Model-based Whitebox fuzzing requires the existence of a file system and the use of an operating system to generate crash reports. Many embedded systems do not use an operating system or have a file system which may not make this approach feasible for a large amount of embedded systems.

Driller is a fuzzing tool that uses dynamic symbolic execution in tandem to find deeper defects [2]. Fuzzing is used to test various sections of code in a program while dynamic symbolic execution is used to generate inputs that is able to pass the complex requirements needed to access the various sections of code. By combining these two approaches, Driller is able to mitigate the weaknesses of both of these approaches: the path explosion of dynamic symbolic execution and the incompleteness of fuzzing. Driller is a white-box fuzzing technique, which may not may it suitable for embedded applications where source code is not available or unattainable.

IoTFuzzer is a fuzzing framework that is designed to discover memory corruption vulnerabilities in IoT devices without requiring access to firmware images [3]. The authors observe that many IoT devices are controlled through an associated mobile app, and that these apps contain a large amount of information about how the app communicates with the embedded system. Using this information, IoTFuzzer is able to discern a way to interface with the embedded application. Since IoTFuzzer interfaces with embedded systems through mobile apps, this approach may not be applicable to embedded systems without such an interface. However, for the IoT space, this is an effective way to leverage an interface tightly coupled with the embedded system. Further, IoTFuzzer does not require that the targeted embedded application have an operating system nor does it require the source code of the embedded system. These features make IoTFuzzer a good approach to fuzzing embedded applications where a mobile app interface exists.

Li et al. developed a program state-based fuzzer, named Steelix, designed to provide a good trade-off between code coverage and execution speed [4]. Steelix uses static analysis and binary code insertion to provide itself comparison progress information. This allows the fuzzing framework to determine where specific comparisons (such as string comparisons) are made in the code, as well as how to mutate the input to bypass these comparisons. Steelix is a grey-box fuzzing approach where information is learned about the program under test through static analysis of the binary. This makes it suitable for fuzzing embedded applications where the source code for the application is unavailable. Steelix uses shared memory to record coverage information and comparison progress. This allows Steelix to glean more information from program runs than simply the output of the program, a good feature for use in embedded fuzzing since feedback from embedded applications may be sparse.

Chipounov et al. developed a program analysis platform called S2E which is designed to facilitate the development of tools for bug finding, reverse engineering of code, and performance profiling [12]. S2E is designed to be scalable through the use of two innovations: selective symbolic execution, a method for minimizing the amount of code that needs to be symbolically executed, and relaxed execution consistency models, a method for intelligently deciding on accuracy/performance trade-offs during program analysis. S2E is able to operate directly on binaries and is able to analyze programs in-vivo, meaning that the platform is able to operate on the program under test while it is running within a real software stack. S2E is built out of an embedded symbolic execution engine, a dynamic binary translator, and a custom virtual machine. S2E provides a number of use-cases for embedded applications. One such application is to use S2E to profile the energy consumption of embedded software. S2E can be provided with a power consumption model and S2E can generate information about the more energy-intensive paths to help developers optimize for power use. Furthermore,

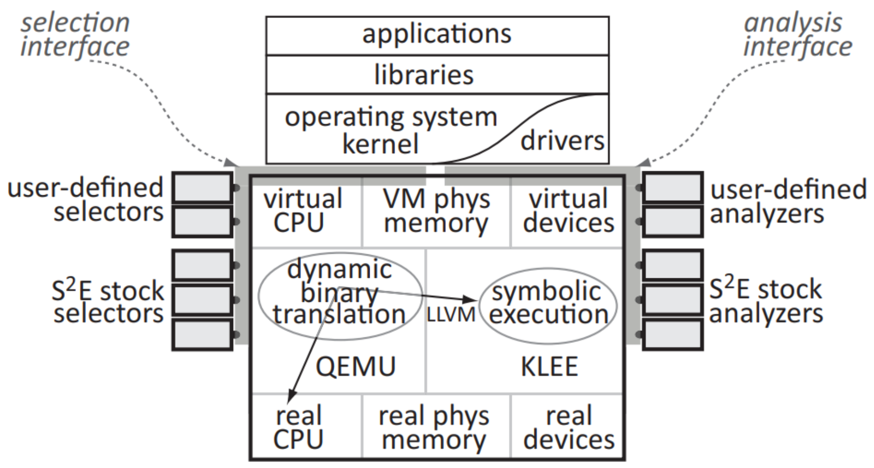

Fig. 1. The S2E architecture.

S2E can also be used to analyze the memory safety of embedded firmware by verifying that along all program paths, memory safety is enforced. S2E may also be used to analyze firmware for privacy leaks. One example the authors provide is the monitoring of the flow of information through a program that operates on credit card information. In such a program, S2E would be able to discern whether any sensitive information is leaked outside of the system in question. Since S2E is able to operate on binaries directly, the technique is able to be applied to embedded systems that are closed source, where no source code is available and the source code may be difficult to reverse engineer. However, S2E does require the use of an operating system to be running on the platform under test to facilitate its functionality, ruling out some embedded systems which may not support an operating system.

BORG (Buffer Over-Read Guard) is a testing tool that utilizes dynamic and symbolic analysis, symbolic execution, and taint propagation to detect buffer overread errors in software [5]. BORG works by identifying buffer accesses that could potentially result in a buffer over-read. Symbolic execution is then used to develop program inputs that can access these areas of code. Since BORG uses static and dynamic analysis to learn information about the program under test, it may be suitable for fuzzing embedded applications, which frequently do not have source code available. However, BORG does require the use of an operating system on the device being tested to run the symbolic execution. This limits BORG's viability for embedded testing to embedded platforms which support an operating system. Since BORG targets one specific type of bug (buffer over-reads), the fuzzer could be used in tandem with other fuzzers to attain a more complete coverage of the different types of bugs that can be discovered in embedded software.

SLF is a fuzzing technique that introduces the ability to generate valid seed inputs by identifying input validation checks and the corresponding input fields that affect such validation checks [6]. These checks are then more specifically categorized by their relation to the input field. Examples of such relations include data structure length checks, object offsets, and relations that are arithmetic in nature. The authors developed a search algorithm designed to apply mutations that are specific to each type of relation, and to generate mutations that will satisfy checks that correspond to multiple relations. Since SFL generates its own seed files, this can be useful when applied to embedded applications where limited prior knowledge is known about the system. Furthermore, SLF does not require that the system under test have an operating system in order to be able to fuzz it. This allows SLF to potentially be able to work with a larger array of embedded systems since many such systems do not come with an operating system.

FIE is a tool developed for symbolic execution and fuzzing of firmware programs written for the MSP430 platform [9]. The authors use the KLEE symbolic execution engine and expand its functionality for use on the aforementioned embedded platform. The authors mention that while much firmware is written in C, existing code analysis tools are ineffective on the embedded platform due to architectural traits that are specific to the low-power nature of the platform. When FIE is used as a fuzzer, it uses information specific to the MSP430 architecture. Such information include the layout of memory, the special registers, and specifics of interrupt handling. As an additional measure to control testing and improve performance, the authors implemented a method to replace reads to peripherals with returned values that are selected from an appropriate range, as well as ignoring writes to peripherals. The authors note that the fuzzing strategy often outperformed the symbolic execution technique when done in a similar timeframe. While symbolic execution can be very slow and resource intensive, in practice many of the firmware programs found on such microcontrollers are relatively simple and can be explored effectively using a symbolic execution approach. However, for any relatively more complex piece of firmware, symbolic execution may encounter the problem of path explosion and diminish in effectiveness. Although FIE is effective for the MSP430 platform, more work would need to be done to adapt such a solution to another particular embedded platform, or to make it functional on any arbitrary embedded platform. This represents a difficult task given the wide array of different embedded platforms that are in use.

Frankenstein is a fuzzing framework designed to fuzz firmware through over-the-air channels [7]. Through firmware emulation, Frankenstein uses firmware dumps to inject fuzzed inputs into the virtual modems of chips. The fuzzing framework has been designed to be fast, allowing it to communicate in real-time with the operating system of the system under test. Given that Frankenstein has been specifically engineered to target firmware, the framework has a lot of applicability to embedded systems that operate with a Bluetooth interface. While this does restrict Frankenstein to embedded systems that support operating systems, many Internet of Things devices are likely to have over-the-air communication functionality. This makes Frankenstein's approach applicable to fuzzing this evolving space. Furthermore, Frankenstein does not require the source code of the program under test to be able to fuzz the target, allowing the framework to be more versatile with the types of embedded systems that are tested with it.

Learn&Fuzz is a fuzzing framework based on statistical machine-learning techniques [10]. The authors use machine learning to automate the generation of valid test inputs given examples of known valid inputs. A new algorithm is presented that guides the test exploration towards interesting parts of the program by leveraging learned input probability distributions. The authors explore the dichotomy of learning, which is the part of the process where information about well-structured inputs is gleaned, and fuzzing, where such well-formed inputs are modified to gain greater program coverage. Given that the fuzzing framework requires known valid test inputs as a seed, Learn&Fuzz may not perform well on many embedded applications that have little to no documentation or such information from a manufacturer. Furthermore, Learn&Fuzz uses an application-agnostic approach which does not utilize any type of coverage feedback to benchmark whether a certain test input has found a deeper path into the program. This may hinder Learn&Fuzz's ability to explore more of the program when testing an embedded application. Lastly, Learn&Fuzz requires that the target device have an operating system in order to be able to fuzz it. Given that many embedded systems lack an operating system, this further reduces Learn&Fuzz's applicability to embedded systems. However, Learn&Fuzz's approach of leveraging machine learning to more intelligently generate test inputs can certainly be useful in the context of mutating test inputs for embedded systems once some valid seed inputs have been aggregated, perhaps by another technique beforehand.

VUzzer is an application-aware fuzzing technique that uses static and dynamic analysis to glean information about the applications being tested [8]. The static and dynamic analysis is used to extract control-flow and data-flow features from the program, and these features are then used as feedback to generate new test inputs. VUzzer uses evolutionary mutation and crossover operations on previous test inputs to guide the search for more interesting test inputs. Since VUzzer does not require any prior knowledge of the application, the fuzzing framework may be able to be used on closed-source embedded applications with limited documentation available. Also, VUzzer uses intelligent techniques to extract information about the program under test. Using dynamic taint analysis, static analysis, and dynamic analysis can allow VUzzer to aggregate much more feedback about a run on an embedded system beyond what the output of the program provides. One limitation with VUzzer is that it requires the aid of an operating system on the platform being tested. For embedded testing, this restricts VUzzer's use to embedded systems that can support an operating system.

Pustogarov et al. introduced a symbolic execution framework, called DrE, for use in testing firmware written for the MSP430 line of microcontrollers [11]. By using the framework develop by the authors, a sequence of sensor inputs can be generated that leads the firmware down a particular code path to a specific point in the software. The authors are able to use their framework on a gesture recognition device and have the device recognize the spoofed input traces as valid input gestures. DrE combines control flow and call graph information that is generated during the execution of the symbolic engine to eliminate infeasible paths of execution and determine the conditions necessary to invoke a desired state in the targeted program. Like other symbolic execution engines, DrE may not perform as well on very complex firmware due to the possibility of path explosion. However, since many samples of firmware for embedded systems are relatively simple, this approach may be effective in the majority of cases in practice. One limitation that DrE faces it that it requires the source code of the program be available for the symbolic execution engine to be able to function. This source code may not be available for a large majority of commercial firmware where the manufacturer may not include this source code. While this project does not specifically employ a fuzzing technique, the interface that is used to inject inputs can be leveraged for future work to develop fuzzers that work with embedded Internet of Things devices to discover vulnerabilities. Using input traces on sensors can be an effective vector to probe for bugs on a large population of firmware programs being used commercially.

7 Future direction

In order to enable more diverse types of program analysis and fuzzing, software re-hosting for embedded systems can be pursued as an area of further research. More work into emulators and simulators for embedded platforms can open the door for various fuzzing tools and techniques that currently only work on other platforms. This type of technology can remove some of the barriers associated with the closed-off nature of embedded systems as well as make the variability of the architectures of embedded systems more approachable.

More work can also be done to explore the use of hybrid testing (such as the type done with the Driller project). This type of testing, where multiple different types of approaches are combined, has seen success with detecting vulnerabilities in other types of software. It stands to reason that a hybrid approach designed specifically with embedded systems in mind may be a powerful tool for discovering defects in them as well.

A more standardized evaluation methodology should also be developed for embedded fuzzing, as well as for fuzzing frameworks in general. Many of the papers examined each had their own evaluation criteria, as so it is difficult to compare the performance of them against one another. A more standarized method for evaluating these tools would demonstrate under which conditions certain techniques and approaches may work more effectively that others.

Since a key challenge with embedded fuzzers is the difficulty of detecting program bugs or failures during test runs, further research should be done to improve the ability of these tools to extract information from various information vectors of the platforms under test. Newly developed fuzzing tools could also explore new channels of potential information such as the physical response of the system (e.g. heat or electromagnetic waves) or the hardware signals themselves.

8 Conclusion

Fuzzing is an automated software testing technique that provides a program under test with generated inputs with the aim of testing the systems response to potentially novel and untested inputs. A fuzzing framework is designed to repeatedly feed a program with such generated inputs in an effort to discover errors in the design of the program. Software fuzzing has been proven to be an effective technique for discovering bugs and defects in software in a large number of domains. Due to the simplicity and performance of fuzzing, it has seen lots of use in finding bugs in many commercial applications. With the advent of the Internet of Things and smart sensor technology, embedded devices are becoming increasingly ubiquitous in daily life and the number of devices we interact with that run embedded software continues to increase. Given that, it is paramount that these devices are secure and safe to use. The application of fuzzing to embedded systems gives us an excellent tool to improve the security of these devices and the software running on them. In this paper, the challenges of embedded fuzz testing has been discussed, an overview of the applicability of some available fuzz testing frameworks to embedded testing has been discussed, and some future direction for further research into embedded fuzzing has been proposed.

References

1. Pham, V., Bohme, M., Roychoudhury, A: Model-Based Whitebox Fuzzing for Program Binaries. ASE 2016
2. Stephens, N., Grosen, J., Salls, Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.,: Driller: Augmenting Fuzzing Through Selective Symbolic Execution.
3. Chen, J., Diao, W., Zhao, Q., Zuo, Q., Zuo, C., Lin, Z., Wang, X., Lau, W., Sun, M., Yang, R., Zhang, K.: IOTFUZZER: Discovering Memory Corruptions in IoT Through App-based Fuzzing. NDSS Symposium 2018
4. Li, Y., Chen, B., Chandramohan, M., Lin, S., Liu, Y., Tiu, A.: Steelix: Program-State Based Binary Fuzzing. ESEC/FSE 2017
5. Neugschwandtner, M., Comparetti, P., Haller, I., Bos, H.: The BORG: Nanoprobing Binaries for Buffer Overreads. CODASPY 2015
6. You, W., Liu, X., Ma, S., Perry, D., Zhang, X., Liang, B.: SLF: Fuzzing without Valid Seed Inputs. ICSE 2019
7. Ruge, J., Classen, J., Gringoli, F., Hollick, M.: Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets.
8. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: Application-aware Evolutionary Fuzzing. NDSS 2017
9. Davidson, D., Moench, B., Jha, S., Ristenpart, T.: FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution. 22nd USENIX Security Symposium
10. Godefroid, P., Peleg, H., Singh, R.: Learn&Fuzz: Machine Learning for Input Fuzzing.
11. Pustogarov, I., Bohme, T., Ristenpart, A, Shmatikov, V.: Model-Based Whitebox Fuzzing for Program Binaries. ASE 2016
12. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. ASPLOS 2011
13. B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," Communications of the ACM, vol. 33, no. 12, pp. 32–44, 1990
14. ClusterFuzz, https://google.github.io/clusterfuzz/
15. Charlie Miller Reveals His Process for Security Research, https://resources.infosecinstitute.com/topic/how-charlie-miller-does-research/
16. Mutiny Fuzzing Framework, https://github.com/Cisco-Talos/mutiny-fuzzer
17. Fuzz testing in Chromium, https://chromium.googlesource.com/chromium
18. OneFuzz framework, https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/
19. Dharma, https://blog.mozilla.org/security/2015/06/29/dharma
20. Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia, A Practical Attack on the MIFARE Classic
21. Ulf Lindqvist and Peter G. Neumann,Inside Risks The Future of the Internet of Things
22. Qian Z, Jiyuan W, Muhammad A, Rohan P, Miryung K, BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction
23. American Fuzzy Lop, https://lcamtuf.coredump.cx/afl/
24. Software and System Security Group, https://www.s3.eurecom.fr/docs/ndss18muench.pdf
25. CERT, "Basic Fuzzing Framework," https://www.cert.org/vulnerability-analysis/tools/bff.cfm.

26. R. Gopinath, B. Bendrissou, B. Mathis, and A. Zeller, “Fuzzing with fast failure feedback,” 2020
27. S. Kim, J. Cho, C. Lee, and T. Shon, “Smart seed selection-based effective black box fuzzing for iiot protocol,” The Journal of Supercomputing, pp. 1–15, 2020.
28. J. D. DeMott, R. J. Enbody, and W. F. Punch, “Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing,” in Proceedings of the Black Hat USA, 2007
29. Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, “Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction,” in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, pp. 722–733.
30. Americaln Fuzzy lop https://lcamtuf.coredump.cx/afl/
31. Marius M, Jan S, Frank K, Aur´elien F, Davide B, What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices
32. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in ACM SIGPLAN notices, vol. 40, 2005
33. N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in ACM SIGPLAN notices, vol. 42, no. 6, 2007.
34. D. L. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. dissertation, Massachusetts Institute of Technology, 2004