Digital Object Identifier (DOI): 10.1109/WIFS49906.2020.9360889

Fuzzing Framework for ESP32 Microcontrollers

Matthias Börsig, Sven Nitzsche, Max Eisele, Roland Gröll, Jürgen Becker*, Ingmar Baumgart
FZI Research Center for Information Technology, Karlsruhe, Germany
Email: {boersig, nitzsche, eisele, groell, baumgart}@fzi.de
*Institute for Information Processing Technology (ITIV), Karlsruhe Institute of Technology (KIT), Germany
Email: becker@kit.edu

***Abstract*—With the increasing popularity of the *Internet of Things* (IoT), security issues in this domain have become a major concern in recent years. In favor of a fast time to market and low cost, security is often neglected during IoT development and little effort has been spent to enhance security tools to support the most common IoT architectures. Therefore, this work investigates fuzzing, an emerging security analysis technique, on the popular ESP32 IoT architecture. Instead of performing fuzzing directly on the target IoT system, we propose a full-system emulator that runs ESP32 firmware images and is able to perform fuzzing several orders of magnitude faster than the actual system. Using this emulator, we were able to fuzz a commercial IoT device with more than 300 requests per second and identify a bug in it within a few minutes. The developed framework can not only be used for discovering security issues in released products, but also for automated fuzzing tests during development.**

I. Introduction

In recent years, the *Internet of Things* (IoT) market has seen a rapid growth with smart sensors and devices in industry and smart home systems. According to Fortune Business Insights, the global IoT market size was $251 billion in 2019 and will reach over $1,463 billion by 2027 [1]. However, in this rapidly evolving market security concerns have been mostly neglected. According to the IoT Threat Report of 2020, 57% of the examined IoT devices were vulnerable to medium- or high-severity attacks [2]. To counteract this situation, existing techniques to discover and avoid security vulnerabilities need to be adopted to work on the embedded platforms commonly used in IoT devices.

An emerging technique, developed for discovering security vulnerabilities, is fuzzing [3]. Fuzzing attempts to crash a program or a system by sending random inputs to it and monitoring the response. A detected crash most likely results from a bug and may indicate a security vulnerability. In general, fuzzing can be separated into two categories depending on whether the target source code is available or not. If it is, then the process is called *whitebox* fuzzing. Otherwise, if only the binary code or a pre-programmed device is available, it is called *blackbox* fuzzing.

Optimally, all possible inputs to a program or system are tested during fuzzing for a comprehensive vulnerability analysis. However, the number of possible inputs is typically too large to be tested in a reasonable amount of time [4]. For this reason, whitebox fuzzing is preferable as the source code and meta-information can help to narrow down the input range and therefore the time required to achieve useful results. On the other hand, many systems and applications are closed source. In such scenarios, blackbox fuzzing can be combined with reasonable assumptions on the functionality of a target application [5], [6]. Alternatively, reverse engineering can be used to analyze the target system prior to fuzzing. This approach is called *greybox* fuzzing [7].

Even though fuzzing techniques have been implemented for some popular microchip architectures [8], there is a growing IoT development platform that has not been considered in any work yet, the ESP32. This is, among others, due to its uncommon processor architecture, *Xtensa*, which is not supported by common tools for security vulnerability analysis. The ESP32 microcontroller was released in 2016 and is built to enable simple development of IoT applications with a short time to market. It offers a number of built-in functions at a very competitive price, including WiFi and Bluetooth communication, which are required for most modern IoT applications. The ESP32 has been sold over 100 million times [9] and is used in several commercial devices already.

This work investigates fuzzing of applications running on ESP32 devices using different approaches. It focuses on achieving high performance in order to not just provide theoretical relevance but also practical usefulness.

II. Our Contribution

In this work, we present a high-performance fuzzing framework for ESP32-based microcontrollers and demonstrate how to use this framework to perform whitebox, greybox and blackbox fuzzing on ESP32 binaries. To the best of our knowledge, no other fuzzing tool supports the ESP32 platform.

The presented framework is based on the emulator QEMU and combines an ESP32 fork[1] (further referred to as *ESP32-QEMU*) with a *Honggfuzz* implementation[2], a QEMU-based fuzzer (further referred to as *QEMU-HONGFUZZ*). We named the resulting ESP32 fuzzing framework *ESP32-QEMU-FUZZ*. The code can be found on GitHub[3].

As proof of concept, we used ESP32-QEMU-FUZZ to fuzz a commercial IoT device and were able to identify a bug in the device within a few minutes.

[1]https://github.com/espressif/qemu
[2]https://github.com/thebabush/honggfuzz-qemu
[3]https://github.com/MaxCamillo/esp32-fuzzing-framework

WIFS'2020, December, 6-11, 2020, New York, USA. 978-1-7281-9930-6/20/$31.00 ©2020 IEEE.

III. Related Work

Muench et al. [8] pointed out challenges in fuzzing embedded systems, like the lack of full-system emulators for fuzzing targets and the unobservability of faults occurring in embedded systems. The advantages of fuzzing embedded systems in an emulator are the transparency of execution and thus the ability to detect faults and collect code coverage.

Voss [10] presented a technique for fuzzing code that is hard to reach under normal test conditions. The fuzzing of functions is enabled by executing the code on the target until it is just before the targeted processing function and then dumping the entire state of the target at this very point. The state is then transferred into an emulator where the generated test data gets injected and the execution is continued. This technique has also been built into the IoT fuzzing framework created by Gui et al. [11]. They developed an additional analysis step that can find relevant parts in the code automatically, prior to the fuzzing process. However, in both works the technique was implemented on top of the *Unicorn* [12] emulator, which does not support the *Xtensa* architecture.

Hertz and Newsham [13] created a project to fuzz QEMU system instances based on American Fuzzy Lop (AFL). Yet, it uses an outdated version of QEMU and can therefore not be combined with the ESP32-QEMU implementation in order to fuzz ESP32 applications. Nevertheless, the project has served as an inspiration for this work.

Bogad and Huber [14] used partial emulation to fuzz firmware images of ESP8266, the predecessor of ESP32. Finally, Li et al. [15] gave a good summary of security vulnerability discovery and in particular on fuzzing, especially coverage-guided fuzzing.

IV. Conception

Fuzzing consists of three steps – input generation, target execution and fault detection – which are explained below.

A. Fault Detection

First of all, the behavior of the ESP32 on memory corruption needs to be investigated. We wrote a test application[4] that processes TCP requests over the WiFi interface and allows triggering the five main causes of memory corruption (stack and heap buffer overflow, null pointer dereference, double free, and unsafe use of `printf`), according to [8], in a controlled manner. The following behavior on the triggered faults could be observed.

Writing outside the boundaries of an allocated buffer is possible on the ESP32. Buffers on the stack will most likely be located in the neighborhood of return addresses while buffers on the heap can also be located beside function pointers. Overwriting one of these will result in a crash of the device if it gets dereferenced. If no important values are overwritten, the ESP32 will not crash and the fault may not be recognized.

With control over the first argument of a `printf` function, it is possible to write to distinct addresses on the stack. If these addresses point to invalid memory locations, the device will crash and the memory corruption is observable.

On dereferencing a null pointer and freeing an already deallocated memory chunk, the ESP32 always crashes. As a result, such causes of memory corruption are always observable.

It is important to note that the fuzzing process does not have to recognize a memory corruption at its first occurrence. During the fuzzing process, a large number of inputs are tested and a present bug, which leads to a memory corruption, will most likely be triggered by various inputs. It is therefore relied on observing a crash of the system to detect memory corruptions.

To improve the detection of those memory corruptions, tools like *AddressSanitizer* or heuristics via emulation could be utilized, as shown in [8].

B. Target Execution with Fuzzing-Hooks

Because the ESP32 is built to serve IoT applications, the input data is normally received through the WiFi interface. Therefore, when fuzzing is performed on the actual device, sending the fuzzing data via WiFi is the most convenient way. This is done by a fuzzing-hook, which needs to repeatedly fetch the fuzzing input data from the *Honggfuzz* interface and send it to the target's network address. It also must verify that the target has sent a response to the request. If no response is received, it is assumed that the target has crashed and the fuzzing-hook must send a fault signal to the fuzzer.

Some targets may require an additional *liveness check* to investigate if the target has not been crashed by the request. Therefore, after sending the request that contains the fuzzing input data, a request that is known to be responded by the target is additionally sent to the device.

C. Coverage-Guided Input Generation

For the input generation, the mutation mechanism from *Honggfuzz* is used. It can either be used by only mutating the provided seed inputs, or by additionally respecting the code coverage information of the target.

1) Compiler-Generated Instrumentation: The ESP32 compiler supports instrumenting the code in order to generate code coverage data, which is then saved to the device memory during runtime. To handle the generated coverage data, the fuzzing-hook needs to download the code coverage data, using a JTAG debugging connection after each tested input, and redirecting it to the fuzzer. The communication flow is shown in figure 1. Unfortunately, only the executed basic blocks are logged by using this method of code instrumentation.

2) Binary Rewriting Instrumentation: To enable coverage-guided greybox fuzzing, the binary code of the application could be modified in order to report executed basic blocks or even compare instruction parameters. Under certain conditions, it is possible to translate the entire binary code back into assembler code and insert instructions that measure the code coverage. Unfortunately, the data fields and instructions in ESP32's binaries are interleaved such that it is impossible to distinguish between them effectively. Also, the dummy bytes

[4]Test application is available at: https://github.com/MaxCamillo/esp32-fuzzing-framework/blob/master/example_esp32_server/main/tcp_server.c

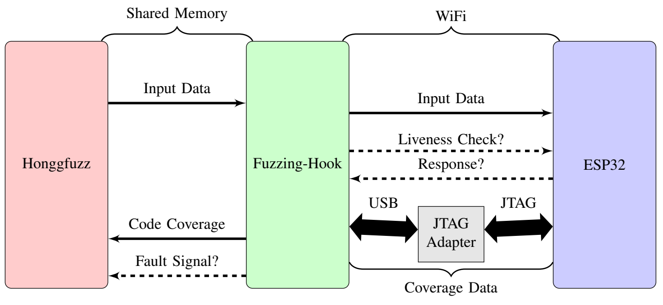

Fig. 1. Coverage-Guided Whitebox Fuzzing for ESP32 applications

for the code alignment make disassembling difficult. It is therefore not possible to translate an entire binary firmware image for the ESP32 back into assembly code, without huge manual efforts.

3) Code Coverage by Emulating: When running an application in an emulator, access to all meta-information of the execution is possible. This transparency can be used to intercept the program counter and the parameters from compare instructions to calculate a fine-grained code coverage. Using a full-system emulator for fuzzing is a highly discussed subject in the community in terms of performance. Some researchers state that the performance of using a full-system emulator is between 2-5 times [16] and up to 10 times [7] worse than the performance of the actual device. In [8], it is shown that an emulator can even be faster than the actual device, especially for embedded devices. Another great advantage of fuzzing in an emulator certainly is the ease of scalability to multiple cores, with multiple instances of the emulator.

V. Implementation

A. Blackbox Fuzzing on ESP32 Applications

Blackbox fuzzing without code coverage consideration on the actual device is the simplest method for fuzzing ESP32 applications. Even though not promising, it is implemented in order to set a performance baseline.

Some services wait for certain symbols at the end of the input data until the processing starts. A header of an HTTP request, for example, ends with two new line characters. If these characters are not received, the processing of the data gets stuck, which would block the whole fuzzing process. Therefore, these characters must be identified and appended to the generated input data by the fuzzing-hook in order to avoid deadlocks.

The crash detection of the target must also be adapted to the actual service. On connection-oriented services, like TCP, it is sufficient to observe whether the connection has been terminated correctly or if a response from the target has been received. For connectionless services, like UDP, more complex methods have to be used to check if the target has been crashed by processing the input data. This can be done with liveness checks after each tested input. However, as shown in [8], liveness checks are not always a reliable option because the embedded systems tend to reboot extremely quickly after crashes and hence the liveness check might even be responded correctly after the target has rebooted. A more sophisticated method for observing crashes on the target is intercepting crash signals with a serial connection to the target. When crashing, the ESP32 always prints a small fault message followed by a reboot message to the serial connection. Intercepting this reboot signal from the serial connection could also be used as a reliable option to observe crashes.

With this method a simple HTTP server application for the ESP32 could be fuzzed with a throughput of about 30-40 requests per second. This fuzzing process is quite inefficient and will probably not trigger any bugs.

B. Whitebox Fuzzing with Compiler Instrumented Code

Considering the code coverage of a tested input for the input generation is necessary in order to increase the efficiency of the fuzzing process. The easiest way to gather the code coverage is to utilize compiler-generated code instrumentation; this is done by recompiling each source file that should be instrumented with the option `--coverage` passed to the compiler. The compiler thereby adds code to each basic block, which counts the number of times it is executed during runtime.

The coverage data is then transferred from the device into *Honggfuzz* by using a JTAG connection. *Honggfuzz* creates a large bitmap in the shared memory in which the addresses of executed basic blocks are stored. Each bit in this bitmap corresponds to one address. Initially, all bits are set to zero. The first time a basic block is executed the corresponding bit is set to one. *Honggfuzz* can detect this bit flip and will store the triggering input as an additional seed in the input folder.

Using such compiler-generated coverage data slows down the fuzzing process by a factor of 10, reaching only about 4 requests per second on fuzzing the simple HTTP server application. Furthermore, only the executed basic blocks are examined by this method and no fine-grained code coverage can be generated. However, the biggest drawback for compiler-generated coverage data is that the source code has to be available.

C. Whitebox Fuzzing with ESP32-QEMU-FUZZ

QEMU-HONGFUZZ examines the code coverage of an emulated user application and directs it to *Honggfuzz*. This modified version of QEMU has been merged into the ESP32-QEMU implementation provided by *Espressif*. Because both implementations are based on nearly the same version of QEMU and have modifications in different areas of the code, merging the two code bases has been straightforward. The result is ESP32-QEMU-FUZZ, a version of QEMU that allows fuzzing of ESP32 applications.

To enable coverage-guided fuzzing, it is required to examine which basic blocks have been executed, as explained in section IV-C. The binary translation engine of QEMU groups instructions into basic blocks, to allow their execution without interruption. This grouping mechanism is reused to determine basic blocks for the fuzzing process. By using an emulator, even more fine-grained code coverage data can be obtained by considering the parameters of compare instructions. Therefore, the two parameters of a compare instruction are intercepted during the translation of the machine instruction within the emulator. The ESP32 architecture provides the functions `strcmp` and `strcasecmp`, which compare two strings either exactly or by ignoring the cases of the chars. All string compare functions are located at fixed addresses in the unmodifiable ROM of the ESP32. This allows to intercept the parameters of the functions when such a function is called within the emulator. The code from *Honggfuzz* needs to be enhanced to additionally handle these parameters. The parameters and the address from which the string compare functions are called can thereby be passed to *Honggfuzz* in the same way as the parameters of the normal compare instructions.

The ESP32-QEMU implementation does not support emulating the integrated WiFi module, so neither does ESP32-QEMU-FUZZ. As a workaround, QEMU offers an Ethernet interface to connect the emulator to the host's network. However, the application needs to be linked against the provided Ethernet driver, which makes this way of communicating to the application exclusively available to whitebox scenarios. In order to redirect the input data from the fuzzer to the host's network address, a fuzzing-hook is created. It is responsible for iteratively fetching the fuzzing input data from *Honggfuzz* and redirect it to the correct network address.

For the fault detection, it is sufficient to intercept the *HALT* interrupt, which is triggered when the ESP32 emulator crashes.

This setup offers a great way for whitebox fuzzing of ESP32 applications. For our TCP test application we could achieve around 80 requests per second using a single thread on a consumer laptop. It can therefore be assumed that the ESP32-QEMU-FUZZ implementation is faster than the actual device, regarding the whole network and data processing. Additionally, fine-grained code coverage of each input is considered for the input generation, which increases the efficiency of the fuzzing process. It is easily possible to scale up this method by using multiple instances. In this case, every instance needs to be bound to its own network interface.

With this whitebox fuzzing implementation in QEMU, it is possible to apply automated fuzz tests within modern continuous integration and continuous delivery development cycles. For this, the application can be compiled separately with the required Ethernet driver and optional stub methods for hardware parts that cannot be emulated.

D. Blackbox and Greybox Fuzzing with ESP32-QEMU-FUZZ

As mentioned before, ESP32-QEMU does not support WiFi. Thus, it is not possible to interact with an application of a blackbox firmware image that uses WiFi in the emulator. Only applications that run without WiFi can be blackbox-fuzzed. Since ESP32-based microcontrollers are often used precisely because of the cheap WiFi module, this is a severe issue. A possibility to solve this issue would be to emulate the WiFi functionality within ESP32-QEMU. For a proper implementation of the WiFi functionality, detailed knowledge of the hardware is required. Unfortunately, the WiFi drivers from the *IoT Development Framework* (IDF) are closed source and there is no documentation of the corresponding WiFi hardware in the manual. A lot of manual reverse engineering would be required in order to examine the internal communication mechanisms. Therefore, we decided to not implement the WiFi functionality, but instead take a greybox approach to fuzz firmware images.

To enable greybox binary fuzzing, the technique from [10] was implemented. It allows to bypass the data input through network interfaces. This is done by saving a state of the actual device after receiving the data and then transferring this state to the emulator. It is sufficient for the fuzzing process to run the code from the beginning of the data processing code until the end of it. Doing so additionally speeds up the fuzzing process, as irrelevant parts are no longer executed. Therefore, the entry and exit points of the data processing have to be found within the firmware. However, almost any random bit string of two and three bytes represents a valid instruction in the *Xtensa* instruction set. As a result, even professional disassembling programs like *IDA Pro* cannot properly disassemble the *Xtensa* code. Hence, finding the entry and exit points requires a lot of manual effort.

One approach to find those regions in the code is to use the step-by-step execution function of *GNU Debugger* (GDB). A deep understanding of the code is needed and can be acquired by using breakpoints and observing the execution. The goal is to find data processing functions and code sections which are good candidates for fuzzing. When the part of the code that is to be fuzzed does not end on a single location, multiple exit points have to be defined. Also, the memory region in which the input data is located must be identified by hand.

When suitable entry and exit points have been found, the state of the target device when reaching the entry point has to be dumped. The state of the ESP32 consists of the values of the 16 registers, the program counter, and the $512\,\mathrm{KiB}$ of static RAM. All of these data can be retrieved by using a JTAG debugging connection and can therefore be dumped easily. To apply this dumped state into the emulator instance, the QEMU implementation had to be modified in order to load the memory image to the appropriate memory region and set all register values accordingly. The input data is then located in the memory of the emulator and can be altered.

To allow a proper continuation of the execution in the emulator, the state must not be loaded before the initialization routines of the firmware. Rather, the firmware must first have run through all important initialization routines to make sure that required modules do work. The point at which the initialization of the operating system is completed is called setup point and has to be identified manually, too. A suitable method to find the setup point of a firmware is to execute the firmware for a few seconds in the emulator and then stop it with the debugger. Usually, the device is now in an idle state and waits for inputs.

On each fuzzing iteration, the input data is overwritten by the input data generated by the fuzzer and the length value is corrected correspondingly. When one of the exit

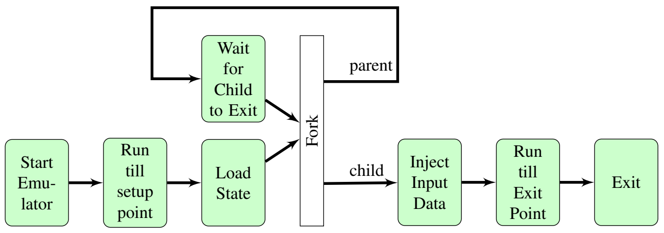

Fig. 2. Fork-Join fuzzing process

TABLE I
COMPARISON OF FUZZING ATTEMPTS ON A INTEL I7-6600U BASED CONSUMER LAPTOP USING ONE CORE

Test	Requests per Second
Blackbox Fuzzing on ESP32 Applications	40
Whitebox Fuzzing with Compiler Instrumented Code	4
Whitebox Fuzzing with ESP32-QEMU-FUZZ	80
Greybox Fuzzing with ESP32-QEMU-FUZZ	320

points is reached, the procedure is started from the beginning. Therefore, each tested input is executed while the emulator is in the exact same state and only the input data differs.

Reloading the whole state of the target after each tested input obviously comes with a big performance penalty. Therefore, a technique called *Fork Server* is implemented.

On *UNIX* systems the *fork* call is realized with the *copy-on-write* policy. As a result, the newly created process shares the memory with the parent process and the whole memory page gets copied if one of the processes executes a write operation. The *fork-join* fuzzing process is illustrated in figure 2.

The behavior of the *fork* call can be used in the fuzzing process to ensure that the same state of the emulator is present in each new fuzzing iteration. Therefore, a child process is created right before the input data gets injected into the emulator's state. The parent process waits for the child process to be finished by using the *join* system call. By considering the return code of the child process, it can be determined whether the child process has terminated with a crash, or because an exit point was reached. Finally, the next child process with newly generated input from the fuzzer can be spawned. Using the *fork-join* mechanism drastically reduces the performance penalty of reloading the device state each iteration.

VI. EVALUATION

A. *Fuzzing the TCP Test Application*

As mentioned in section IV-A, we used a TCP test application to verify the ESP32-QEMU-FUZZ implementation. With coverage-guided whitebox fuzzing it was possible to find input combinations to trigger all implemented bugs within a few hours. The achieved requests per second for each test is shown in Table I.

B. *Greybox Fuzzing the LIFX Mini*

To prove the effectiveness of the implemented fuzzing method, it is tested against a commercial product. The *LIFX Mini* is a smart light bulb, which contains an ESP32 as control unit. It is a typical IoT consumer device that gets controlled via WiFi. For this, a smartphone app is offered that can manage multiple smart light bulbs at once. As expected for commercial devices, no source code is available.

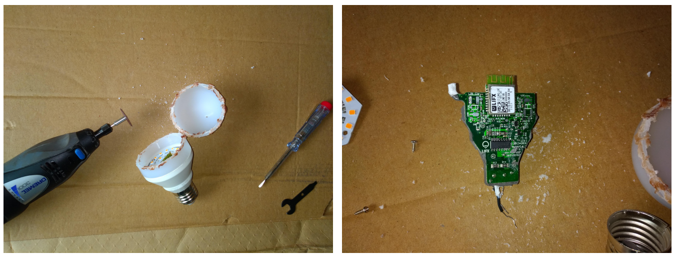

Fig. 3. A disassembled LIFX Mini smart light bulb.

1) Preparation of the Target: At the first examination of the target device, we discovered that the JTAG port has been deactivated. The permanent deactivation of the JTAG port is a security feature of the ESP32, which cannot be undone. Therefore, the firmware of the light bulb needed to be transferred to an ESP32 development board, in order to be able to use the JTAG connection. The 4 MB flash memory of the ESP32, which contains the firmware, can be dumped via a serial connection. To reach the pins for the serial connection, the light bulb had to be opened, as shown in figure 3.

Next, the firmware is dumped using the *esptool* provided by IDF. The firmware image of about 750 kB is then written to the flash memory of the ESP32 developer board, which is connected to a JTAG adapter. This allows to set breakpoints at arbitrary locations and read the state of the device.

2) Fuzzing the Initial Configuration Process: For the initial configuration, the light bulb operates its own WiFi access point. The smartphone app makes the phone to join this access point and to establish a secure TLS channel to the light bulb on TCP port number 56700. The credentials of the user's WiFi access point are then exchanged over the secure channel. After the initial configuration, the light bulb joins the user's WiFi access point and further control commands are transferred via this network. The focus of the first fuzzing approach of the light bulb will be on this initial configuration process.

At first, the entry and exit points had to be found using GDB. Then, the dumped device state and metadata had to be provided to the ESP32 fuzzing framework. As a result, several crashes could be observed within a few minutes. Two different ways were found to force the device into infinite looping, whereby the device is not usable for about 30 seconds. After this amount of time, the device reboots itself. This bug could be exploited for *Denial of Service* (DoS) attacks in order to prevent the device from being used. Furthermore, an input was found that caused the device to crash and restart. However, the restart seems to be triggered by a built-in reboot command. So far, no memory corruptions and fault signals could be observed by the manual analysis of the crash.

After about 45 minutes, the fuzzing process no longer explored new code areas. Even running the fuzzing process for another 72 hours did not result in any new code coverage. It can therefore be assumed that most of the accessible code parts have been covered in the first 45 minutes. This would prove the effectiveness of the fine-grained code coverage gathering.

3) Fuzzing on Open TCP Port: The light bulb also offers an HTTP server at TCP port number 80 after initialization. After a few minutes of fuzzing, a null pointer exception could be found. The exception results from an unsafe use of the `strchr` library function, which is located in the unmodifiable ROM of the ESP32. If the character is not contained in the string, this function returns a pointer to a demanded character within a string or a null pointer. In this case, the returned pointer was not verified but dereferenced, and thereby caused the exception.

As mentioned before, exploitation of bugs on the *Xtensa* architecture has not yet been investigated heavily and no exploitation of null pointer exceptions has been reported yet. However, several possibilities have been found on other architectures to exploit these kinds of exceptions. Null pointer dereferences are listed in the *2019 CWE Top 25 Most Dangerous Software Errors* [17] in the 14th place. This bug should therefore be considered as a potential serious security vulnerability. The discovering of this bug shows the effectiveness of the coverage guided greybox fuzzing.

VII. Limitations and Future Work

The modified version of QEMU that was developed in this work is limited to ESP32. It could be extended to all embedded systems that are supported by QEMU. This would allow to utilize the implemented methods for a lot more microcontroller architectures. However, since QEMU does not support all available architectures, the ability to fully emulate arbitrary firmware images, as mentioned in [8], still remains an open problem.

VIII. Conclusion

For some IoT development platforms fuzzing techniques have been developed in the past. Prior to this work, however, there were no published tools for IoT devices built upon the ESP32 platform. Different techniques for fuzzing ESP32 applications in various scenarios have been implemented and evaluated in this work. The two methods that are particularly effective rely on fuzzing in an emulator instead of running on the actual device. For instance, whitebox fuzzing enables automated continuous testing during the application development process. This method of fuzzing ESP32 applications fits seamlessly into a modern agile application development process. The greybox fuzzing option of the framework provides a powerful tool for security analysts. It can be utilized to perform fuzzing on parts of the firmware that appear vulnerable to a security analyst. This method was tested against a commercial device and could find bugs in little time. The null pointer dereference error, which could be found in the tested commercial device, is a potential security vulnerability. Currently, no method to exploit this type of error on the ESP32 has been published yet. But with the increasing popularity of the ESP32, it is just a matter of time until the financial benefit for attackers is high enough to focus on this kind of exploits.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research within the framework of the project KASTEL_SKI in the Competence Center for Applied Security Technology (KASTEL).

References

[1] Fortune Business Insights, "Internet of Things (IoT) Market Size, Share and Industry Analysis By Platform (Device Management, Application Management, Network Management), By Software & Services (Software Solution, Services), By End-Use Industry (BFSI, Retail, Governments, Healthcare, Others) And Regional Forecast, 2020-2027," 2020.

[2] Unit 42, "2020 unit 42 iot threat report," https://unit42.paloaltonetworks.com/iot-threat-report-2020, 2020.

[3] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[4] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, 2011.

[5] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, 2015.

[6] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[7] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1099–1114.

[8] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.

[9] Espressif, "Espressif achieves the 100-million target for iot chip shipments," https://www.espressif.com/en/media_overview/news/espressif-achieves-100-million-target-iot-chip-shipments, 2018.

[10] N. Voss, "afl-unicorn: Part 2 fuzzing the 'unfuzzable'," https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5, 11 2017.

[11] Z. Gui, H. Shu, F. Kang, and X. Xiong, "Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution," *IEEE Access*, vol. 8, pp. 29 826–29 841, 2020.

[12] N. A. Quynh and D. H. Vu, "Unicorn-the ultimate cpu emulator," https://www.unicorn-engine.org/, 2015.

[13] J. Hertz and T. Newsham, "Triforceafl," https://github.com/nccgroup/TriforceAFL, 6 2016.

[14] K. Bogad and M. Huber, "Harzer roller: Linker-based instrumentation for enhanced embedded security testing," in *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ser. ROOTS'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3375894.3375897

[15] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.

[16] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37 302–37 313, 2018.

[17] M. Corporation, "2019 cwe top 25 most dangerous software errors," https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, 2019.